Article

International Journal of HIGH PERFORMANCE COMPUTING APPLICATIONS

The International Journal of High Performance Computing Applications
2015, Vol. 29(4) 403–421

DOI: 10.1177/1094342014532297
hpc.sagepub.com
SAGE

Silent error detection in numerical time-stepping schemes

Austin R Benson[1,2], Sven Schmit[1] and Robert Schreiber[2]

Abstract

Errors due to hardware or low-level software problems, if detected, can be fixed by various schemes, such as recomputation from a checkpoint. *Silent errors* are errors in application state that have escaped low-level error detection. At extreme scale, where machines can perform astronomically many operations per second, silent errors threaten the validity of computed results.

We propose a new paradigm for detecting silent errors at the application level. Our central idea is to frequently compare computed values to those provided by a cheap checking computation, and to build error detectors based on the difference between the two output sequences. Numerical analysis provides us with usable checking computations for the solution of initial-value problems in ODEs and PDEs, arguably the most common problems in computational science. Here, we provide, optimize, and test methods based on Runge–Kutta and linear multistep methods for ODEs, and on implicit and explicit finite difference schemes for PDEs. We take the heat equation and Navier–Stokes equations as examples. In tests with artificially injected errors, this approach effectively detects almost all meaningful errors, without significant slowdown.

Keywords

Silent errors, resilience, Runge-Kutta, linear multi-step methods, heat equation, initial-value problems

1. Silent errors and checking schemes

1.1 Silent errors are worrisome

Computational scientists are concerned about silent errors in exascale computing. Silent errors are perturbations to application state that may lead to a failure such as a bad final solution (Snir et al., 2013). These errors may arise from a bit flip, a firmware bug, data races, and other causes. Several authors (Cappello et al., 2009; Dongarra et al., 2011; Snir et al., 2013) have discussed the sources and the frequency of silent errors.

Why the current concern? An exaflop machine will be able to do on the order of 10^{23} operations per day, and will have on the order of 10^{17} bytes of memory (Dongarra et al., 2011). And in order to achieve very aggressive energy efficiency and performance targets, machine architects are pushing envelopes: with near-threshold-voltage logic, with new memory and storage technologies, and with photonic communication. Consumer quality hardware may already suffer errors at the personal-computer scale once per year (Nightingale et al., 2011), and cost precludes really significant hardening of the hardware in supercomputers. Thus, the scale of systems makes such errors quite likely. Indeed, some high performance systems today already suffer from silent errors at a troublesome rate (Shi et al., 2009).

Furthermore, the practice of checkpoint/restart is not well suited for exascale applications in general, and in particular to silent errors. Disk-based checkpointing is prohibitively expensive with expected error rates in exascale applications. While in-memory checkpointing is an option (Zheng et al., 2012), the difficulty in detecting silent errors is more worrisome. Without knowing an error has occurred, we do not know that a restart is necessary. This makes the checkpoint/restart paradigm a weaker strategy for guarding against silent errors.

1.2 Algorithmic responses to silent errors

The numerical algorithms community has already looked at error vulnerability. It is well known that

[1]Institute for Computational and Mathematical Engineering, Stanford University, CA, USA
[2]HP Labs, Palo Alto, CA, USA

Corresponding author:
Austin R Benson, Institute for Computational and Mathematical Engineering, Huang Building, 475 Via Ortega, Suite 060 (Bottom level), Stanford University, Stanford, CA 94305-4042, USA.
Email: arbenson@stanford.edu

many errors do not cause failures. Other errors lead to an obvious application failure. Silent errors are more worrisome, because they can cause unsuspected erroneous outputs. Our goal is to make these errors non-silent.

It has been argued that with extra care, convergent iterations are inherently self-correcting; for example, a resilient version of GMRES is proposed in Hoemmen and Heroux (2011). Other empirical studies have shown, however, that iterative methods are sometimes vulnerable to errors (Bronevetsky and de Supinski, 2008; Casas et al., 2012). And in a study of a minimization approach to Hartree–Fock ground state calculation, Van Dam et al. found that 'it is insufficient to rely on the algorithmic properties of the Hartree–Fock method to correct all the possible bit-flips and resulting data corruption' (Van Dam et al., 2013). Sufficiently big errors can be fatal to these algorithms.

Minimization and equation-solving (in which the data defining the function or equation are assumed to be incorruptible) is an easy case, since the residual of the current approximate solution almost surely does not lie. But in many cases in computational science, a time-dependent initial-value problem is solved. In these cases, any perturbation to the computed solution puts that solution onto a permanently erroneous track. We therefore take the view that error detection is a fundamental issue, and that errors, detected as soon as they occur, can then be handled by an appropriate correction scheme.

Certain common kernels have been fortified with error detectors. For example, checksum methods have been used for matrix multiplication (Huang and Abraham, 1984), high performance LU factorization (Du et al., 2012), and checking the integrity of data replicated on multiple compute nodes (Van Dam et al., 2013). This latter paper also monitored a number of theoretical invariants of the algorithm. Orthogonality of a matrix can be checked by multiplying by its transpose, for example. Conservation laws can be monitored, where available, as a check for error. These monitoring approaches were found to be useful, but fallible; they are not a comprehensive safety net.

1.3 Our approach

In this paper, we propose a very general, low-cost error detection approach that applies to iterative computations in general, and to the solution of initial-value problems for ODEs and PDEs in particular. Our central idea is to compare the solution given by a primary, or *base*, time-stepping scheme to the solution given by an auxiliary *checking* scheme, and to do this every time step. The two schemes use the same input data, all of it computed by the base scheme: thus the auxiliary solution is used only locally, at each time step, to check for errors. This approach has compelling advantages over a straightforward duplication of the computation: it is cheaper, and it can detect problems that duplication cannot.

Error is a constant in scientific computing. Even with no bugs or failures, we have modeling error, truncation error, and roundoff error. To deal with truncation error, schemes of the kind we are proposing have long been employed for automatic step-size control in ODE solvers (Fehlberg, 1969). Similar a posteriori error estimators are used for mesh adaptation in PDE solvers (Berger and Oliger, 1984). The idea is to make a numerical method introspective, aware of, and watchful for errors. Our contribution is to extend these powerful schemes so that they can be used to detect errors due to a misbehaving computing system as well.

Suitable checking schemes are available in the most common setting: the solution of initial-value problems in ODEs and PDEs. A full description of the general approach, including the key question of how we trigger notification of an error, is given in the next section. We describe specific checking schemes for Runge–Kutta and linear multistep methods (LMMs) in Section 3.1 and for finite difference methods for the heat equation in Section 3.2. We discuss the error detector, and an approach to controlling and optimizing it, in Section 4.

In tests with artificially injected errors, we measure the impact of errors by how much they impact the solution. We quantify this idea in Section 5.1, and we then show through numerical experiments that our detection scheme effectively catches errors that have a significant impact on the solution.

2. Outline of general method

Suppose there are two iterative methods to solve a problem, a base method, $\mathcal{B}$, and an auxiliary checking method, $\mathcal{A}$. One would use $\mathcal{B}$ in a computing environment with no errors. Desirable properties of $\mathcal{B}$ are therefore accuracy and stability. A suitable auxiliary $\mathcal{A}$ solves the same problem: its output can be compared to that of $\mathcal{B}$, and it can be used at each iteration, using the same input data as $\mathcal{B}$. The key idea is that the norm of the difference between the results provided by $\mathcal{A}$ and $\mathcal{B}$ is an estimator of the magnitude of the difference at the current step between $\mathcal{B}$ and an error-free solution. This suggests that $\mathcal{A}$ should have accuracy comparable to (or even better than) $\mathcal{B}$. For efficiency, we want $\mathcal{A}$ to be fast when used as a check on $\mathcal{B}$, possibly by reusing some of the computations, communications, and input data of $\mathcal{B}$. Since we do not use $\mathcal{A}$ in a closed-loop setting (i.e. $\mathcal{A}$ does not use its own results as input at each step), stability is not an issue for $\mathcal{A}$. This gives us useful freedom in choosing auxiliary schemes.

The two schemes produce sequences of values $\{A_i\}$ and $\{B_i\}$ in the same normed vector space. For any

norm or seminorm of the difference, we have a scalar sequence, $D_i = \| A_i - B_i \|$. We may choose to use more than one such metric, so in general, D_i may be a vector.

Our methods employ, at each step n, a window into the sequence $(D_{n-d}, \ldots, D_n)$, as data for an error detection function $E(D_{n-d}, \ldots, D_n)$ that decides whether or not to raise the flag for an error. The error detector typically employs one or more measures of size, or, more powerfully, measures of anomaly, to the value D_n in the context of its recent values $D_{n-d}, \ldots, D_{n-1}$. In general, then, a method includes base and checking computation schemes, a vector of difference measures, and an error detection criterion.

2.1 Choosing an auxiliary scheme

Let us first consider the simplest case, and show why it does not help us; this motivates the search for better auxiliary schemes.

Consider $\mathcal{A}$ to be the same as $\mathcal{B}$, that is, $\mathcal{A}$ just repeats the computation. The error detector simply flags an error whenever $\mathcal{B}$ and $\mathcal{A}$ differ, or differ by more than a small multiple of machine precision. Here $\mathcal{A}$ is not fast (it costs the same as $\mathcal{B}$). Moreover, it will not catch certain errors: if the input data for the step are corrupted (after successful previous steps write these data to memory), the two computations produce identical, incorrect results. The same is true if a computation unit fails in a repeatable manner (a stuck-at fault, for example), or a communicated value corrupted by the network is used by both schemes.

A better checking scheme is one that reuses some of the computation and communication of the base scheme (for efficiency), but is different in a way that makes the two schemes disagree unless there are no errors anywhere in the algorithm.

Two examples, covered in detail below, are embedded Runge–Kutta schemes, in which $\mathcal{A}$ reuses evaluations of the derivative function that are needed for $\mathcal{B}$, and paired LMMs, in which saved and new values of the solution and its time derivative are combined in two different ways to estimate the solution at the next time step. Notably, for stiff ODEs and second-order parabolic PDEs, stability mandates the use of implicit base schemes, with their attendant algebraic system to solve at each time step, but explicit schemes prove to be effective, inexpensive auxiliaries.

2.2 The detector

The norm of the difference, $D_i = \| A_i - B_i \|$, is an obvious candidate for error-checking. What complicates this is that the size of D_i can vary over orders of magnitude in the error-free case as the solution changes. Thus, a hard threshold is ineffective for error detection. Our view is that a sudden change in the sequence $\{D_i\}$ better indicates that an error is present. We examine this empirically in Section 5.

At the core of the detector there will be comparisons of some scalar indicator quantities to some thresholds. How should these thresholds be chosen? We take the following view. With any sort of error detector, we can have false positives and false negatives, and there is an intrinsic tradeoff between their rates: a low threshold boosts the rate of false positives but misses few true errors; a high threshold reduces the false-positive rate (FPR) at the expense of more missed errors. In our setting, a more pressing issue is that the indicator quantities can vary by orders of magnitude for different applications. To make matters worse, even within a single instance, these indicator quantities can vary by equal amounts. Therefore, we have to ensure that the detector is very flexible in finding standard levels for the indicators and even able to do so locally. More detail is given in Section 4.

2.3 What to do if an error is flagged

What do we expect an application to do if an error is detected? This is not the topic of our work, but we feel it is important to give an idea of a general scheme.

Before implementing any error recovery scheme for an iterative method, there are two important questions to ask when a flag is raised to indicate an error:

1. On what iteration could the error have occurred?
2. What data or computations were affected by the fault?

We then intend to redo the failed steps (Question 1) by redoing all potentially failed computations (Question 2).

How far back must we go? In our numerical experiments, we see that in quite a few cases the time step *after* the one in which the fault occurred causes the error flag to be raised. For example, a small error in a derivative evaluation in an LMM may produce a large error in a few time steps, since the derivative evaluation gets re-used for several iterations. Thus, we first have to establish which iterations may be erroneous and which ones we still trust. This depends very much on the application, as the following example illustrates.

Suppose the base and checking schemes use the solutions at the previous two time steps to compute the solution at the next. Further suppose that this pair of schemes sometimes flags an error one iteration after the faulty step. Consider now that our procedure flags iteration 5, signalling there might be an error. We cannot trust the solution at step 4, but we can trust step 3 because, were it faulty, we would have seen a flag at iteration 3 or 4. Hence, we go back and restart the computation of iteration 4, using the stored data from iterations 2 and 3.

We hope and expect that silent errors will be rare. Even with a small FPR, there will be more false positives than true positives. It is important to not get stuck and redo (correct) computations over and over again when they are incorrectly flagged. In order to avoid this we propose a taxonomy of possibilities:

1. The error may have been caused by a transient fault. On retry, if the fault does not recur, we will likely succeed, with no error flag.
2. The error may have been caused by a permanent fault that causes erratic, irreproducible, and random errors. This may be the case if a single processor core is faulty and produces different results when executing the same code.
3. If data in memory have been corrupted, silently, we may discover this with our scheme. If we redo the failed steps starting from the same corrupted in-memory data, we expect an identical outcome, with the error flag raised on the retry.
4. If some hardware or software component has failed in a 'hard' way, meaning it consistently produces incorrect results, we expect an identical outcome again (cf. point 2 above, where the fault is permanent, but the outcome is *not* identical), with the error flag raised on the retry.
5. The error may be expected if, for example, the algorithm sacrifices correctness in rare scenarios for speed (Rinard, 2013). If the algorithm fails randomly, but rarely (as is the case in Rinard, 2013), re-computation will likely provide the correct answer. If the failure is deterministic, then a different algorithm will be needed.
6. There may be no error at all; it may be that the problem's local difficulty causes the error flag to trigger, with the current value of the time step, in the error-free case. Again, we expect an identical result on retry.

As stated above, the first response to a flagged error is to go back to an iteration that was computed with no error flags, or possibly one iteration further back, and redo the subsequent iterations, including the error checks. We also want to check to see whether this recomputation produces the same result as it did initially.

If retry succeeds with no error flag, we likely have discovered that an error of type 1 or 5 occurred. We can report it, and continue.

If retry fails, but with a significantly different result to that of the first try, it is likely that a component of the platform has become unreliable; we need to change it. This is an error of type 2. Note that case 4 is different: the fault is permanent but the output is consistently incorrect.

If retry fails with the same computed result as initially, there is ambiguity: the cause may be any of 3, 4, 5, or 6 above. Absent a way to tell, there is a problem.

Our approach is compatible with systems that protect memory contents from loss and corruption. All machines have a basic error detection and correction system for memory. These can be augmented with additional protection and redundancy, as in the Global View Resilience Project (Fujita et al., 2013). In the case of a reproducible flagged error, such a scheme can be invoked to test the memory contents and see if they have been corrupted, to rule our an error of type 3, or correct it if one has occurred.

If such a test detects no corruption of the input data to the failed step, then how can we distinguish between errors of type 4 and 6? Here we could go back to the origins of this approach, and redo the computation with a smaller step; perhaps half the current step. If the error is of type 6, this approach will, after a few reductions, fix the problem. But if the problem remains after repeated step-size reductions, we would suspect an error of type 4.

3. Applications

3.1 ODE solvers

Consider a first-order ODE initial-value problem:

$$\frac{d}{dt}u(t) = f(t, u(t)), \quad u(0) = u_0 \tag{1}$$

Suppose that we are using the explicit midpoint Runge–Kutta scheme as a base scheme $\mathcal{B}$ to compute $u^{\mathcal{B}}_{n+1} \approx u(t_{n+1})$ in equation (1):

$$k^{\mathcal{B}}_1 = f(t_n, u^{\mathcal{B}}_n) \tag{2a}$$

$$u^{\mathcal{B}}_{n+1} = u^{\mathcal{B}}_n + hf\left(t_n + \frac{1}{2}h, u^{\mathcal{B}}_n + \frac{1}{2}hk^{\mathcal{B}}_1\right) \tag{2b}$$

The local truncation error (LTE) of this scheme is $\mathcal{O}(h^3)$. We note that $f(t_n, u^{\mathcal{B}}_n)$ is the central computation in Euler's method, which has LTE $\mathcal{O}(h^2)$, and we use this to construct an auxiliary scheme $\mathcal{A}$,

$$u^{\mathcal{A}}_{n+1} = u^{\mathcal{B}}_n + hk^{\mathcal{B}}_1$$

An example difference computation is $D_{n+1} = \| u^{\mathcal{B}}_{n+1} - u^{\mathcal{A}}_{n+1} \|_\infty$. By re-using $u^{\mathcal{B}}_N$ and $k^{\mathcal{B}}_1$, $\mathcal{A}$ provides a cheap approximation to the solution. The midpoint and Euler schemes are an *embedded Runge–Kutta pair* (Dormand and Prince, 1980); these form the basis of adaptive step-size methods. In general, we can use any embedded Runge–Kutta pair in the $\mathcal{A}/\mathcal{B}$ formulation. A common, accurate scheme is the RKF45 scheme due to Fehlberg (Fehlberg, 1969).

Figure 1 illustrates how errors lead to jumps in the difference between $\mathcal{A}$ and $\mathcal{B}$ using this particular scheme for the Van der Pol equation:

$$u''(t) - b(1 - u(t)^2)u'(t) + u(t) = 0 \tag{3}$$

whose rapid changes in derivatives make this a challenging case.

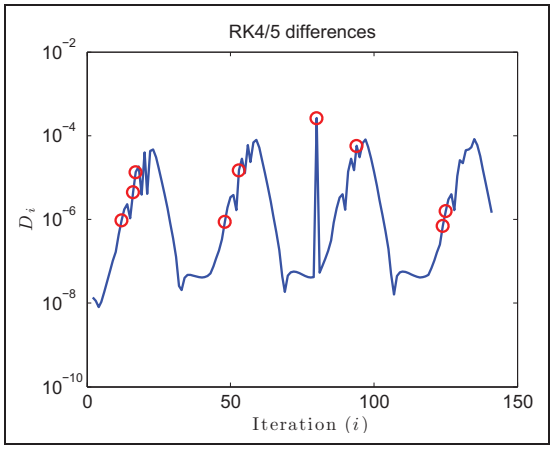

Figure 1. Difference between RK5 ($\mathcal{B}$) and RK4 ($\mathcal{A}$) over time for the Van der Pol equation with $b = 2$ and initial conditions $u(0) = 1$ and $u'(0) = 0$. An artificial error is injected at the 80th iteration, which results in the spike in D_{80}. The red circles indicate iterations that are predicted to be erroneous by our detection scheme; see Section 4.

In Section 5, we show in more detail that errors in the evaluation of equation (2a) or equation (2b) can be effectively detected by Runge–Kutta-based $\mathcal{A}/\mathcal{B}$ schemes. Moveover, we also show that errors in the evaluation of f can be detected just as effectively, even if this wrong computation is used by both methods. Note that in Runge–Kutta methods, the last solution is the initial condition for advancing to the next time step. This memoryless property makes it difficult to detect changes in $u_n^{\mathcal{B}}$. We discuss this matter and provide experiments in Section 5.2.

LMMs are also amenable to our framework. An Adams–Bashforth LMM (AB-LMM) of order $p \geq 1$ computes $u_{n+1}^{\mathcal{B}} \approx u(t_{n+1})$ by

$$u_{n+1}^{\mathcal{B}} = u_n^{\mathcal{B}} + \sum_{i=n-p+1}^{n} h\alpha_{p,i} f(t_i, u_i^{\mathcal{B}})$$

such that the LTE is $\mathcal{O}(h^{p+1})$. Suppose that $\mathcal{B}$ is a pth-order AB-LMM, $p \geq 2$. One choice of $\mathcal{A}$ is the AB-LMM of order $p-1$, which reuses the same data, stores no additional data, and performs no additional evaluation of f. An alternative $\mathcal{A}$ is an LMM of order p that interpolates at (possibly multiple) $u_k^{\mathcal{B}}$ for $k<n$. However, additional memory is needed to store solutions at prior time steps. In order to compare AB-LMM to Runge–Kutta methods, we will consider the $(p-1,\ p)$ AB-LMM pairs in our experiments in Section 5.

Implicit numerical schemes are preferred for stiff ODEs. For example, an Adams–Moulton LMM (AM-LMM) of order p defines u_{n+1} implicitly, as the solution to the system of equations

$$u_{n+1}^{\mathcal{B}} = u_n^{\mathcal{B}} + \sum_{i=n-p+2}^{n+1} h\beta_{p,i} f(t_i, u_i^{\mathcal{B}})$$

Its LTE is $\mathcal{O}(h^{p+1})$. Suppose that the base scheme $\mathcal{B}$ is an AM-LMM. The computationally expensive part of the method is solving the (possibly nonlinear) equation for u_{n+1}. A lower-order AM-LMM will require a different solve and is a less attractive choice for $\mathcal{A}$ (it will not be fast compared to $\mathcal{B}$). Instead, we can use an AB-LMM for $\mathcal{A}$,

$$u_{n+1}^{\mathcal{A}} = u_n^{\mathcal{B}} + \sum_{i=n-p+1}^{n} h\alpha_{p,i} f(t_i, u_i^{\mathcal{B}})$$

AB-LMM is an explicit method, but the starting value, $u_n^{\mathcal{B}}$, and the prior function evaluations, $\{f(t_i, u_i^{\mathcal{B}})\}_{n-p+1 \leq i \leq n}$, have been computed by the implicit AM-LMM. Thus, use of an AB-LMM as $\mathcal{A}$ does not suffer from the instability of explicit methods used on stiff ODEs. Note that we can employ any implicit LMM as a base scheme, including, for example, the backward differentiation formulas.

Finally, an explicit/implicit pair of LMMs is sometimes used in a predictor–corrector fashion. Here, the implicit LMM's equations are solved by a truncated fixed-point iteration in which the explicit scheme generates a first iterate. In this instance, the auxiliary scheme (the predictor) is already a part of the solution mechanism for the base scheme (the corrector), so it comes at no extra cost.

3.2 PDE solvers

Due to the large variety of PDE solvers, we do not have a one-size-fits-all solution. For time-dependent PDEs, a *method of lines* discretization in space results in a system of ODEs, and the ODE methods described above can be employed. Here instead we consider finite difference schemes for PDEs.

To make this idea concrete, we will describe an $\mathcal{A}/\mathcal{B}$ formulation for the heat equation. A more detailed example (the incompressible Navier–Stokes equations) is provided in Section 5.6.

For a model problem, consider the nonhomogeneous heat equation

$$\begin{aligned} u_t &= ku_{xx} + q(x,t), \quad k>0 \\ u(x,0) &= v(x) \end{aligned} \tag{4}$$

with homogeneous Dirichlet boundary conditions. Suppose that $\mathcal{B}$ and $\mathcal{A}$ are the backward and forward Euler schemes. Both methods have LTEs of $\mathcal{O}\left((\Delta x)^2\right)$ in space and $\mathcal{O}(\Delta t)$ in time. At each time step, $\mathcal{B}$ solves a linear system, while $\mathcal{A}$ computes a matrix–vector product. Thus, we expect $\mathcal{A}$ to be faster. Moreover, in the distributed memory setting, $\mathcal{A}$ requires no communication other than what is done in scheme $\mathcal{B}$, if $\mathcal{B}$ uses an iterative solver.

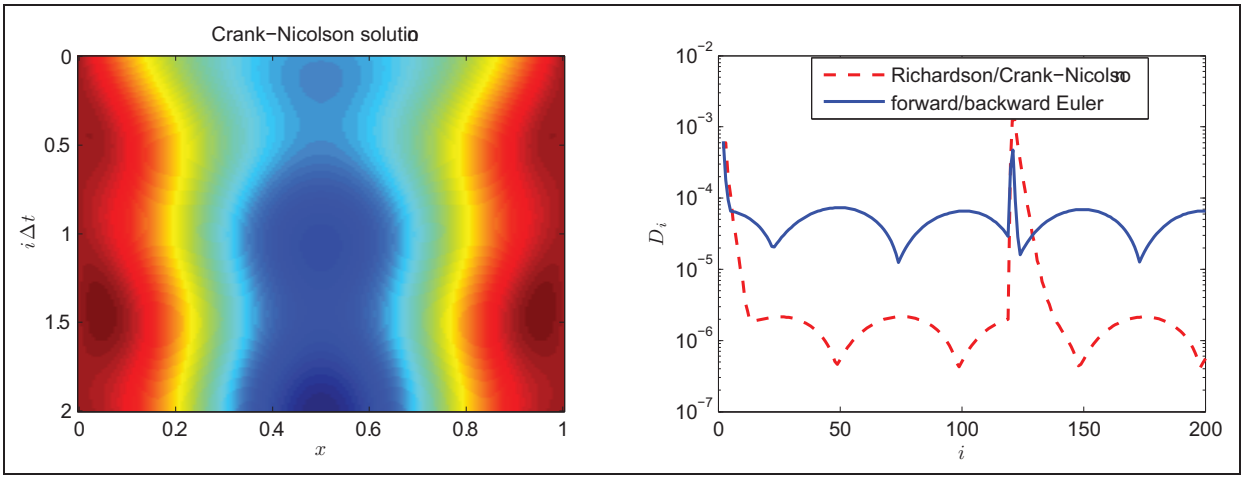

Figure 2. Solution and difference sequence for equation (4) with $k = 1/100$, $q(x,t) = 0.1(\sin(2\pi t) + \cos(2\pi x))$, $v(x) = x(x-1)$, $\Delta x = 1/160$, and $\Delta t = 1/100$. The difference function is $D_i = \| A_i - B_i \|_\infty$. A fault is injected at the 120th time step by multiplying the 40th component of the right-hand side of the linear solves used by Crank–Nicolson and backward Euler by 0.995. The solution appears normal, but the difference sequence indicates an anomaly at the 120th time step.

An alternative $\mathcal{B}$ is the Crank–Nicolson scheme, which is implicit and has LTEs of $\mathcal{O}((\Delta x)^2)$ in space and $\mathcal{O}((\Delta t)^2)$ in time. Forward Euler is a candidate for $\mathcal{A}$, but we desire an explicit method with the same LTEs as Crank–Nicolson. The Richardson scheme (also known as the leapfrog scheme) is such a method, and it uses a centered difference in time and space. While the Richardson scheme reuses the computations from Crank–Nicolson, the centered difference in time requires the solution at the two previous time steps. We refer to Strikwerda (2007, Section 6.3) for a discussion of all of these methods.

Figure 2 plots the solution to the heat equation and the sequence of differences $\{D_i = \| A_i - B_i \|_\infty\}$ for a particular problem instance. An error occurs in the 120th iteration and is exhibited by a spike in the sequence of differences. In Section 5.3, we thoroughly examine how effective the Richardson/Crank–Nicolson (R/CN) and forward/backward Euler (FE/BE) $\mathcal{A}/\mathcal{B}$ formulations are in detecting errors.

3.3 Extrapolation

For some base schemes $\mathcal{B}$, the choice of a related auxiliary $\mathcal{A}$ may not be obvious. But extrapolation is always available, in the form of an LMM in which all the β terms are zero. The order-one version is simply

$$A_i = 2B_{i-1} - B_{i-2}$$

Although useless as basic solvers, extrapolation methods are suitable for error detection. They are very cheap, and can have acceptable error characteristics, even though more custom-tailored auxiliary schemes, if available, will probably be more effective. In cases where no such custom-tailored auxiliary schemes are readily available, or they are too complicated to implement or too expensive to compute, we can employ extrapolation to extend our approach to many more settings.

We investigate the usefulness of extrapolation as part of the $\mathcal{A}/\mathcal{B}$ scheme used for the Navier–Stokes equations in Section 5.6.

4. Error detection

In this section we outline a practical implementation of the error detection function, E. Thereafter, we briefly discuss the performance penalty incurred by employing our error detection scheme. Throughout this section, we consider a scalar error metric D, for example the sup-norm of the difference between A and B. The main challenge for the detection and focus of this section is that the scale of the variations in D is unknown, and can be time-varying, and hence it is important that E can handle errors independent of scale, and adapt to the local structure. Furthermore, we also need our detection scheme to be easy to compute, as this has to be done every iteration.

To find iterations with errors, we use two indicator variables derived from the sequence of differences D_{n+1}:

$$J_{n+1} = \frac{D_{n+1} - D_n}{D_n} \tag{5}$$

$$V_{n+1} = \frac{\mathrm{Var}(D_{n-p+1}, \ldots, D_{n+1})}{\mathrm{Var}(D_{n-p}, \ldots, D_n)} \tag{6}$$

J_{n+1} and V_{n+1} measures the jump in the sequence and V_n measures a change in variance. By using relative

changes, we are less prone to changes in magnitude. A large value for either indicator signals that an error has occurred. The integer p adjusts the window size; in our experiments, $p = 10$.

We flag an error only when both indicators exceed their current thresholds. We show in Section 5.4 that the two-indicator strategy improves the sensitivity to actual errors for a fixed FPR.

We use a closed-loop mechanism to tune the thresholds. We increase a threshold by a factor $\Gamma > 1$ every time the indicator is above its threshold and decrease a threshold by a factor $\gamma < 1$ every time the indicator is below its threshold. If both indicators are above their respective thresholds, we flag an error and rely on an error handler as discussed in Section 2.3. The overall idea is in Algorithm 1.

The closed-loop tuning procedure forces thresholds to be only slightly above typical values, and adapt quickly to changes. In practice, this reduces the probability of false negatives, without causing many false positives. Because the thresholds are decreased every time they are not violated, they do get violated at some point, in which case we increase the threshold again. But as long as this does not happen to both at the same time, we do not trigger an error. Hence, we get this adaptivity almost for free.

For our experiments in Section 5, we use $\Gamma = 1.4$, $\gamma = 0.95$, and $p = 10$. However, the detector's performance is not sensitive to these choices. We choose γ to be close to one so as to reduce the FPR. While on the one hand it is desirable that the performance of the detector does not depend on the choices for Γ and γ, this also means that we cannot easily decide on the proper tradeoff between sensitivity and specificity. This should not be too much of a concern, as we show that our algorithm is rather accurate in detecting errors, and that with a very small FPR.

Algorithm 1 Pseudocode for the error detection algorithm. We use adaptive error thresholding to keep the FPR and false-negative rate low.

```
procedure RESILIENT ALGORITHM
  Initialize thresholds τ_J and τ_V.
  Initialize increase parameter Γ>1.
  Initialize decrease parameter γ<1.
  while n<N do
    B_{n+1} = BaseMethod()
    A_{n+1} = AuxiliaryMethod()
    D_{n+1} = ‖ B_{n+1} − A_{n+1} ‖
    // Compute indicators
    J_n = (D_{n+1} − D_n)/D_n
    V_n = (Var(D_{n−p+1}, ... D_{n+1}))/
    (Var(D_{n−p}, ..., D_n))
    // Check for errors
    if J_n>τ_J and V_n>τ_V then
      FlagError()
      Move backward: n = n − x
    else
      UPDATETHRESHOLD(J_n, τ_J)
      UPDATETHRESHOLD(V_n, τ_V)
      Move forward: n = n + 1
    end if
  end while
end procedure

procedure UPDATETHRESHOLD(t,τ)
  if t>τ then
    τ = Γτ
  else
    τ = γτ
  end if
end procedure
```

4.1 Performance

There is some performance overhead when employing Algorithm 1. Although it is important to understand and quantify the effect on computation time, the overhead depends heavily on the application. Below, we give a general characterization of the overhead and discuss the performance for the applications in Section 5.

There are three factors that play a role in the performance. First, we need to compute the auxiliary solution at each time step. In the methods described in Section 3, the cost of this extra computation is small. For example, in the heat equation, backward Euler solves a linear system while forward Euler only computes a matrix–vector product. Second, at each time step, we need to compute the difference D_n and the indicators J_n and V_n. These are negligible compared to the cost of the base method. Third, every time an error is flagged, we have to redo one (or more, depending on the algorithm) iteration. This is the dominant factor in the performance overhead. Suppose we have to redo k iterations for every flagged error. Then, the extra computational cost is approximated by k times the FPR (true positives do not count as a performance penalty). In the applications in Section 5, k is two or three, and the FPR is below 10%. Thus, in the worst case in our experiments, there is around a 30% overhead. In some cases, the FPR is less than 0.1% (see Section 5.3), in which case the performance overhead is negligible.

4.2 Discussion

Our method relies on two assumptions:

1. Methods $\mathcal{B}$ and $\mathcal{A}$ produce approximately the same result in the error-free case; that is, they are accurate.
2. Changes in the solution are not excessively rapid, so the behavior of the difference sequence is predictable in the error-free case.

When either of these assumptions is violated, we expect problems. For example, a discontinuity in the derivative f in equation (1) may cause consecutive iterations to vary wildly, and an error will often be predicted. And indeed we may have detected an error: not one caused by an unstable platform, but rather one due to underresolution. It is thus inherent to our approach that when the solution changes rapidly, we may see false positives.

We note that sometimes an error is flagged one step *after* the iteration the error occurred. In some methods, this occurs due to error propagation. In other cases, the reason is more subtle, and we explore delayed error detection for the heat equation in Section 5.5. In our experiments, we did not see cases where the error gets flagged later than one iteration after it has occurred. Usually, the difference between $\mathcal{A}$ and $\mathcal{B}$ methods diminishes rapidly following the error (see Figures 1 and 2).

Many more sophisticated statistical tools are available in time series analysis for outlier and peak detection (Hamilton, 1994; Lin et al., 2003). However, algorithms for peak detection are not a focus of this paper, and the simple and fast approach outlined above performs well for several practical examples in Section 5.

5. Numerical experiments

In this section, we evaluate the performance of several $\mathcal{A}/\mathcal{B}$ formulations on a variety of problems. We begin by outlining how we inject artificial faults into computations. Next, we elaborate on the particular problem instances and how the detector performs. All experiments used Matlab R2013a. The data and code for our experiments are available online at http://stanford.edu/~arbenson/silent.html.

5.1 Fault injection and LTE-normalized error

In our experiments, we inject faults by corrupting important computations or data, such as the result of a function evaluation. We do not corrupt internal data structures or program logic. The reason for this choice is that low-level errors in the program will likely either *not* be silent or will have a similar effect to corrupting computations or data. They will also be harder to control. Our main interest lies in demonstrating that our approach works whenever an error is significant, no matter how it came about. Therefore, we study a more artificial setting that allows for more fine-grained control of the magnitude of the difference between the two methods. Evaluating our method on a bit flip that causes a change of the order 2^{50} is not very useful, as it should always be detected, and similarly an error on the order of 2^{-50} is also not interesting, as it does not impact the solution.

The three ways we inject faults are as follows:

1. Corrupt an evaluation of f, the derivative function in equation (1), or an evaluation of q, the source term in equation (4).
2. Corrupt the right-hand side when solving a system of linear equations *before* the solver is used.
3. Corrupt a previous solution from the solver (most of our solvers use the solution from the previous time step).

By 'corrupt', we mean multiply a single component of a vector or matrix by some amount. In our experiments, we conduct many trials and multiply by a normally distributed random variable with mean 1 and problem-dependent variance, σ^2.

Suppose that a fault is injected at iteration $n-1$. In order to measure the impact of a fault on the computed solution relative to the ordinary truncation errors of the numerical method, we use the value

$$L_n = \frac{\| B_n - \hat{B}_n \|}{\| \hat{B}_n - \hat{A}_n \|}$$

where $\hat{B}_n$ and $\hat{A}_n$ are the outputs of $\mathcal{B}$ and $\mathcal{A}$ when no fault is injected. The numerator measures the magnitude of the impact of the fault on the base solution, and the denominator is an estimate of the LTE. We call L_n the *LTE-normalized error*.

The advantage of using a normalized quantity is that we can more easily compare performance for different time step sizes, or between different applications. Also, by measuring the LTE-normalized error, the type of error introduced in the experiments becomes less important. Instead, we are able to see how detection varies with the error's impact on the solution. A small LTE-normalized error means that the error has relatively little influence on the solution while a large LTE-normalized error means that the error has a large influence on the solution. In the subsequent sections, we show that our detector effectively catches large LTE-normalized errors but has difficulty catching small LTE-normalized errors. In practice, this is a desirable property: errors that have a stronger impact on the solution are more easily detected. Furthermore, it seems unreasonable to demand that errors of the order of LTE are detected.

5.2 Van der Pol equation

Our first set of experiments uses the Van der Pol equation (equation (3)). We will vary the damping parameter b in our experiments but fix the initial conditions and time interval:

$$u(0) = 1, \quad u'(0) = 0, \quad t \in [0, T] = [0, 14]$$

The Van der Pol equation has rapid changes in derivatives, which makes it a difficult test problem for our

error detection scheme. Increasing b stiffens the problem and induces more rapid changes in derivatives.

We test four $\mathcal{A}/\mathcal{B}$ schemes: Runge–Kutta 4/5 (RK45), Runge–Kutta 2/3 (RK23), Adams–Bashforth 4/5 (AB45), and Adams–Bashforth 2/3 (AB23). In the first set of experiments, we corrupt one component of the derivative evaluations at some time step ($\sigma^2 = 1 \times 10^{-1}$). Runge–Kutta uses four (RK23) or six (RK45) derivative evaluations per step, and the error corrupts one of these evaluations. Adams–Bashforth uses one derivative evaluation per step, so the time step determines the corrupted function evaluation. The erroneous time step, corrupted derivative component, and erroneous evaluation in Runge–Kutta are chosen uniformly at random. We use 2000 trials, where a trial consists of an ODE solve at times $0, h, 2h, \ldots, T$. One error is introduced per trial. Finally, we use two values of the damping parameter, $b \in \{2, 3\}$. For $b = 2$, the step sizes are $1/10$ and $1/20$ for the Runge–Kutta and Adams–Bashforth methods, respectively. For $b = 3$, the step sizes are $1/15$ and $1/35$.

Figure 3 shows the true-positive rate (TPR) as a function of the LTE-normalized error. The TPR is the proportion of artificially injected errors detected by the detection scheme. We use a kernel regression with a Gaussian kernel to fit the TPR to the LTE-normalized error. Each plot shows the detection rate for both (1) detection at the time step of the fault and (2) detection at the time step of the fault or the step after. In all cases, we see the trend that large LTE-normalized errors are easily detected while small LTE-normalized errors are more difficult to catch. Contrary to RK45 and RK23, AB45 and AB23 detect many errors the step after the error occurs. This is not entirely surprising. Runge–Kutta methods use the erroneous derivative evaluation once to advance a time step, while Adams–Bashforth methods reuse the erroneous computation at the time step of the fault *and* in following steps. Thus, there is more opportunity for the $\mathcal{B}$ and $\mathcal{A}$ schemes to disagree in Adams–Bashforth methods. Finally we note that, in general, the higher-order schemes (RK45 and AB45) exhibit slightly better performance than the lower-order schemes (RK23 and AB23).

In the second set of experiments, we corrupt previous time step data stored in memory. Error-correcting codes in memory hardware provides a low-level check for faults that corrupt data in memory, and applications can supplement these with other, perhaps stronger, protections, at some cost; but it is interesting to find whether our approach can detect changes in stored data independently.

For Runge–Kutta, the relevant stored data is the solution computed at the last time step. For an LMM, the stored state is a set of several solutions and derivatives at previous time steps.

Figure 4 shows the error detection effectiveness of Runge–Kutta and LMM-based schemes. We see that the Runge–Kutta $\mathcal{A}/\mathcal{B}$ schemes have difficulty detecting the errors. At each step of any Runge–Kutta, the previous solution is the initial condition for advancing to the next time step. Thus, the difference computation $D_n = \| u_n^{\mathcal{B}} - u_n^{\mathcal{A}} \|$ does not necessarily seem out of the ordinary; D_n is the correct difference for the wrong problem. The change in initial conditions can cause D_n to be significantly larger than D_{n-1}, so the detection rates are still modest. With a multistep method, on the other hand, the previous solution and derivative evaluations need to be correct for D_n to be the correct difference. Thus, AB45 and AB23 detect these errors effectively; the TPR is quite high when the error is the result of corrupting stored solution or derivative data.

These results illustrate an advantage of LMMs compared to one-step methods. It could be argued that a checksum could be used to detect changes to data stored in memory, and that these could be used in conjunction with one-step $\mathcal{A}/\mathcal{B}$ schemes for error detection: one can perform a check on the memory content at each step, before accepting the solution at the next step. But these schemes cannot detect data corruption due to a bug that stores an incorrect value. Thus, it is important to be able to detect memory data corruption at the application program, and multistep schemes appear to do this effectively.

5.3 Heat equation

We consider the heat equation (equation (4)) with homogeneous Dirichlet boundary conditions, for $x \in [0, 1]$, $t \in [0, T]$. The $\mathcal{A}/\mathcal{B}$ formulations are the R/CN and FE/BE schemes described in Section 3.2. Table 1 describes the three configurations of the heat

Table 1. Three configurations of the heat equation.

Configuration	$q(x,t)$	$v(x)$	k	T	Δx	Δt
1	$xe^{-t/2}$	$4x(x-1)(x-2)$	$\frac{1}{100}$	2	$\frac{1}{100}$	$\frac{1}{60}, \frac{1}{100}, \frac{1}{140}$
2	$\frac{1-\sqrt{1-4(t-t^2)}}{2-2t}$	$6\left\|x-\frac{1}{2}\right\|-3$	$\frac{1}{1000}$	1	$\frac{1}{200}$	$\frac{1}{100}, \frac{1}{200}, \frac{1}{400}$
3	$0.1(\sin(2\pi t) + \cos(2\pi x))$	$x(x-1)$	$\frac{1}{100}$	2	$\frac{1}{160}$	$\frac{1}{100}, \frac{1}{160}, \frac{1}{200}$

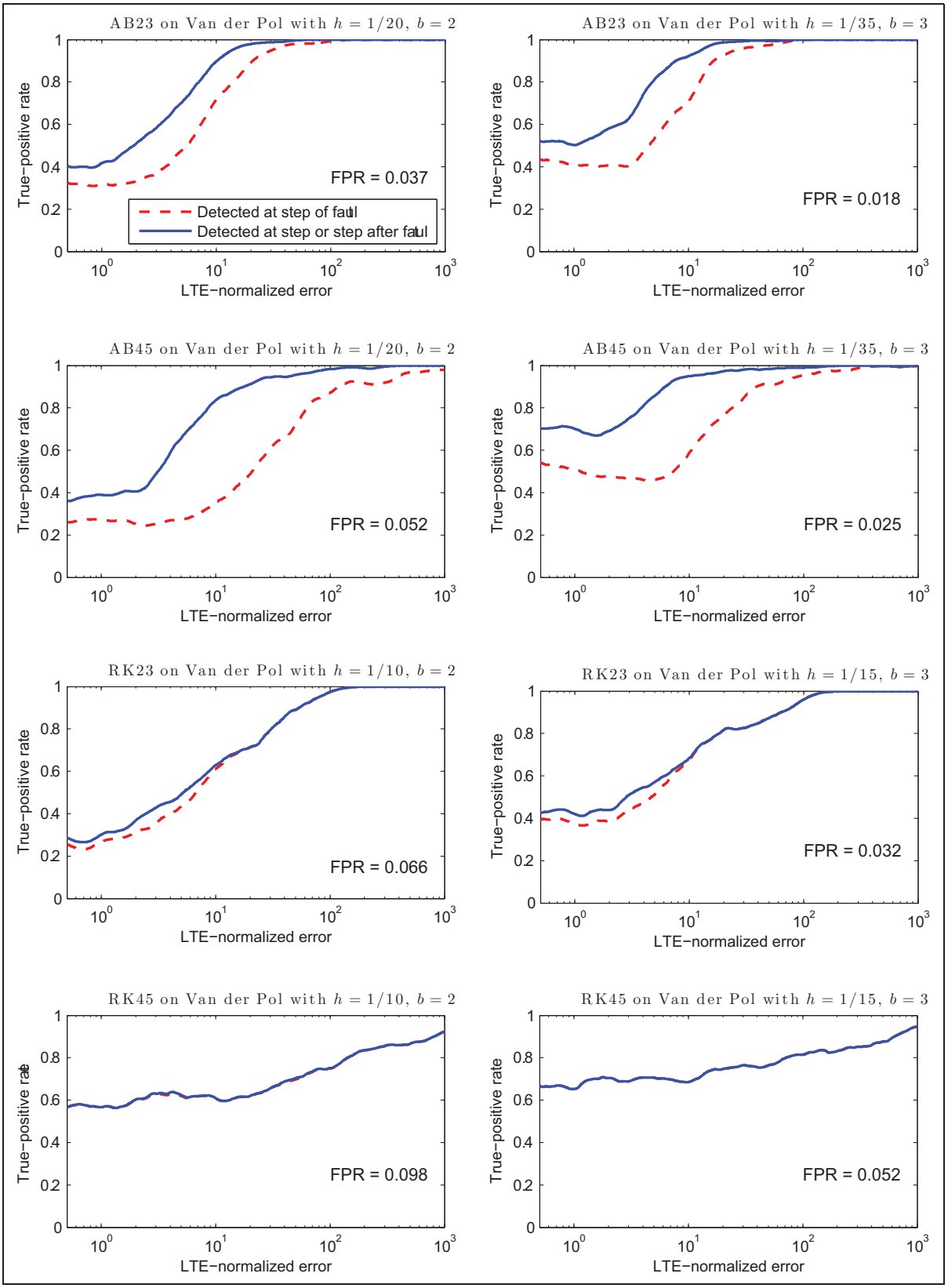

Figure 3. Detector performance with RK45, RK23, AB45, and AB23 $\mathcal{A}/\mathcal{B}$ schemes on the Van der Pol equation. We corrupt a single derivative evaluation at the current time step by multiplying one component of the evaluation by a normal random variable with mean 1 and variance 1×10^{-1}. The same sequence of corruption amounts (values of the random variable) was used for each plot. Kernel regression with a Gaussian kernel was used to compute the curves.

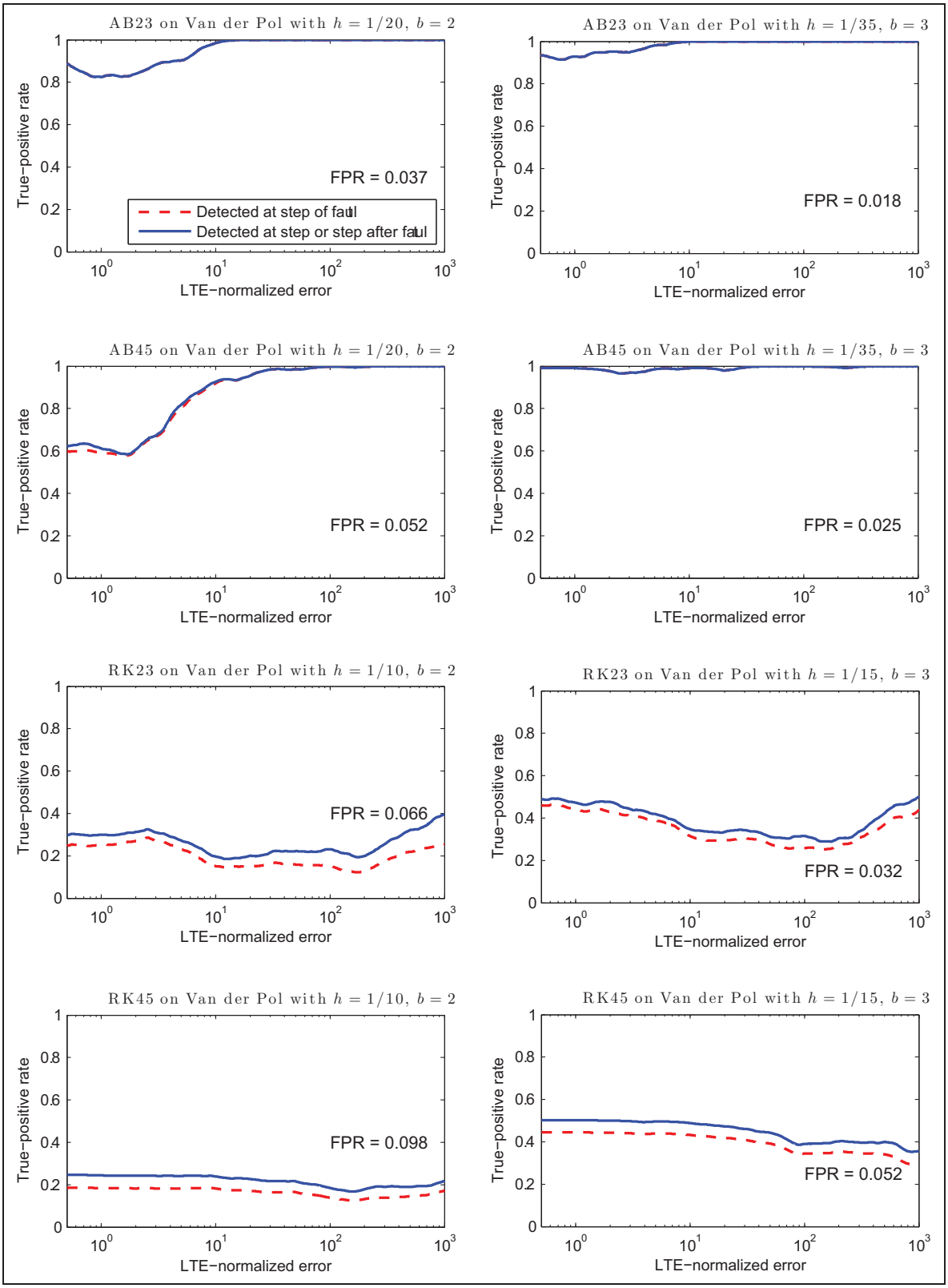

Figure 4. Detector performance with RK45, RK23, AB45, and AB23 $\mathcal{A}/\mathcal{B}$ schemes on the Van der Pol equation. We corrupt the solution from the last time step (or a previous derivative evaluation in AB23 and AB34) by multiplying one component of the vector by a normal random variable with mean 1 and variance 1×10^{-1}. The corrupted data is stored in memory in the program. The same sequence of corruption amounts (values of the random variable) were used for each plot. Kernel regression with a Gaussian kernel was used to compute the curves.

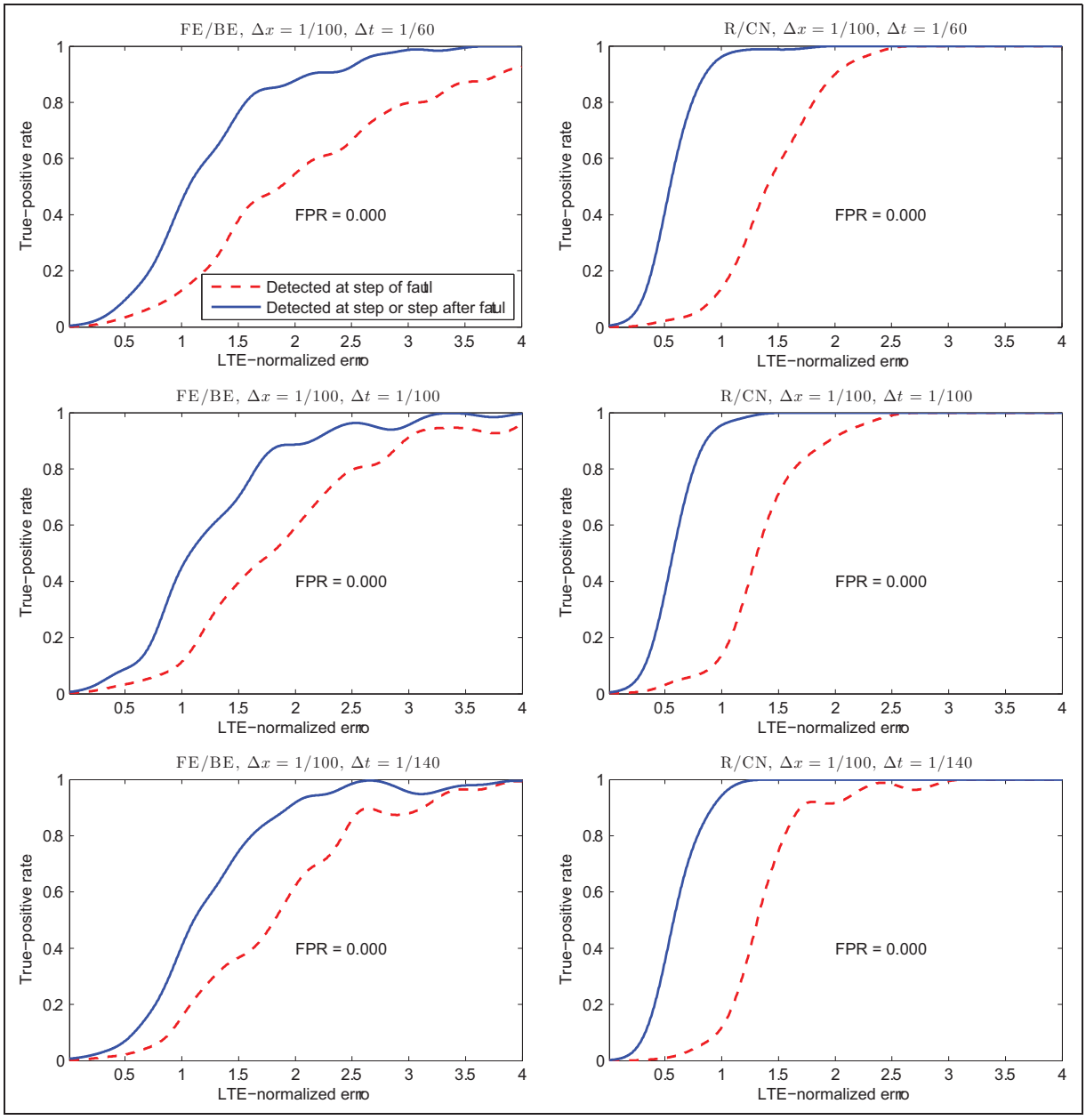

Figure 5. Detection results for heat equation under Configuration 1 with $dt = 1/60$, $1/100$, and $1/140$. Faults are injected by multiplying the source term q by a normally distributed random variable with mean 1 and variance 1×10^{-3} (R/CN) or 1×10^{-1} (FE/BE).

equation used in our experiments. For each configuration, we perform experiments with different time steps.

We consider a trial to be one call to the heat equation solver, which finds a numerical solution at spatial points $0, \Delta x, 2\Delta x, \ldots, 1$ and temporal points $0, \Delta t, 2\Delta t, \ldots, T$. We inject one fault per trial at a uniformly random step. For Configuration 1, we corrupt a single component of the function evaluation of q ($\sigma^2 = 1\times10^{-3}$ for R/CN and $\sigma^2 = 1\times10^{-1}$ for FE/BE). For Configuration 2, we corrupt a single component on the right-hand side of the implicit schemes' linear systems ($\sigma^2 = 1\times10^{-6}$ for R/CN and $\sigma^2 = 5\times10^{-5}$ for FE/BE). For Configuration 3, we corrupt a single component of the previous solution vector ($\sigma^2 = 1\times10^{-6}$ for R/CN and $\sigma^2 = 1\times10^{-4}$ for FE/BE). The variances were chosen in order to generate errors with LTE-normalized error near one. If the variances were much larger, nearly all errors would be detected and we would not see the relationship between TPR and LTE-normalized error. The variances used were smaller for R/CN than for FE/BE because the same type of corruption is more easily detected by

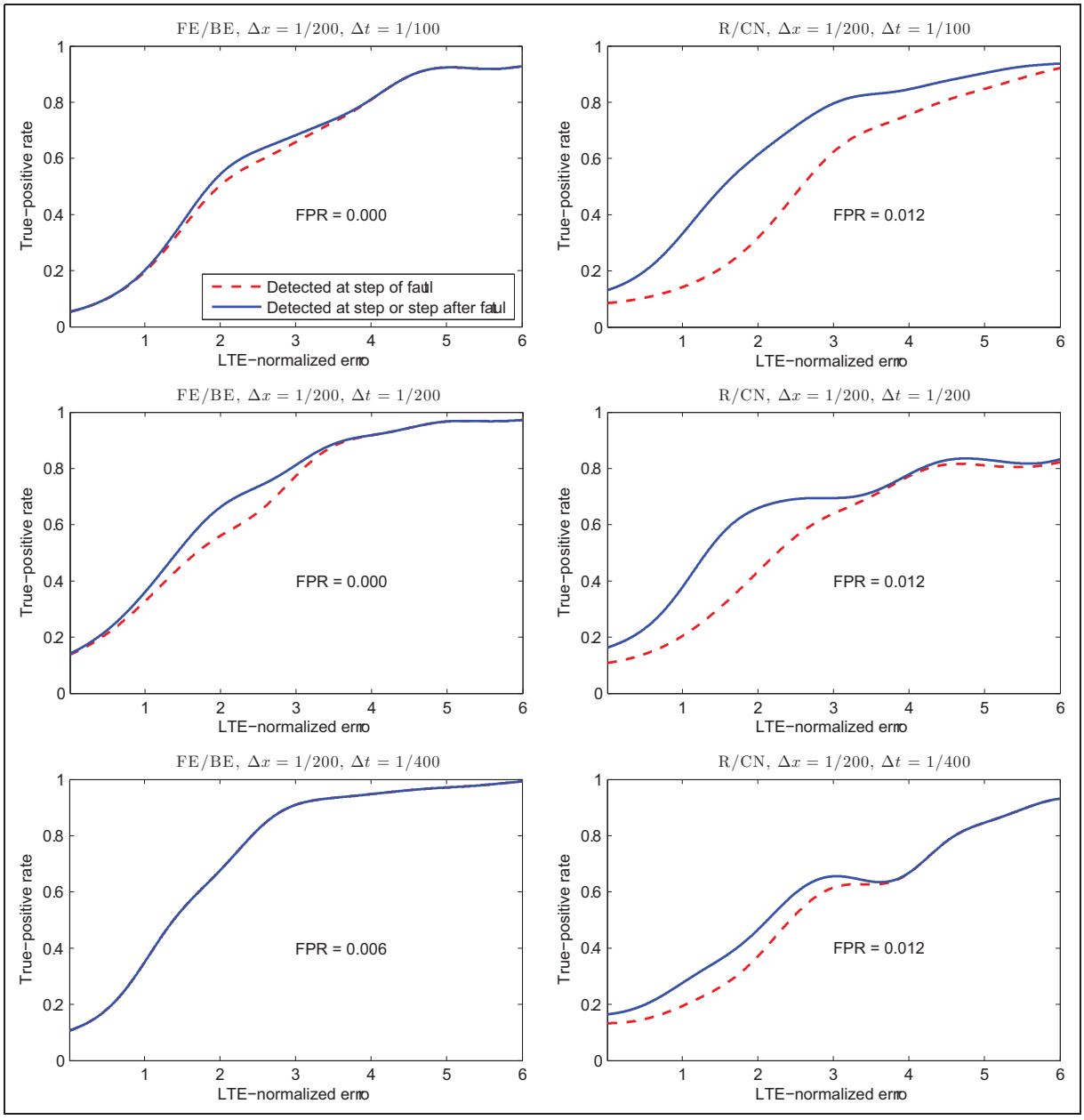

Figure 6. Detection results for heat equation under Configuration 2 with $dt = 1/100$, $1/200$, and $1/400$. Faults are injected by multiplying a single random component of the right-hand side of each linear equation solve by a normally distributed random variable with mean 1 and variance 1×10^{-6} (R/CN) or 5×10^{-5} (FE/BE).

R/CN. This agrees with the case for the ODE solvers in Section 5.2, where higher-order $\mathcal{A}/\mathcal{B}$ schemes had better detection rates.

Figures 5, 6, and 7 plot TPR as a function of the LTE-normalized error. The results illustrate several important features of the detection scheme. First, errors with a large impact on the solution (large LTE-normalized error) are much more easily detected than errors with a small impact (small LTE-normalized error). Second, checking for an error one step after the fault occurs can significantly improve detection (see especially Figure 5). In Section 5.5, we explore why this is true. Third, the FPR is small. In many cases, no false positives are produced. The largest FPR was only 1.2%. Fourth, we can detect several types of errors. Finally, decreasing the time step either improves detection rates or keeps the detection rates the same.

We note that the adaptive thresholding, described in Section 4, does not allow for a tradeoff of better TPR at the cost of a larger FPR or vice versa. In a sense, adaptive thresholding approximately finds the sweet spot where any anomaly that can be detected is

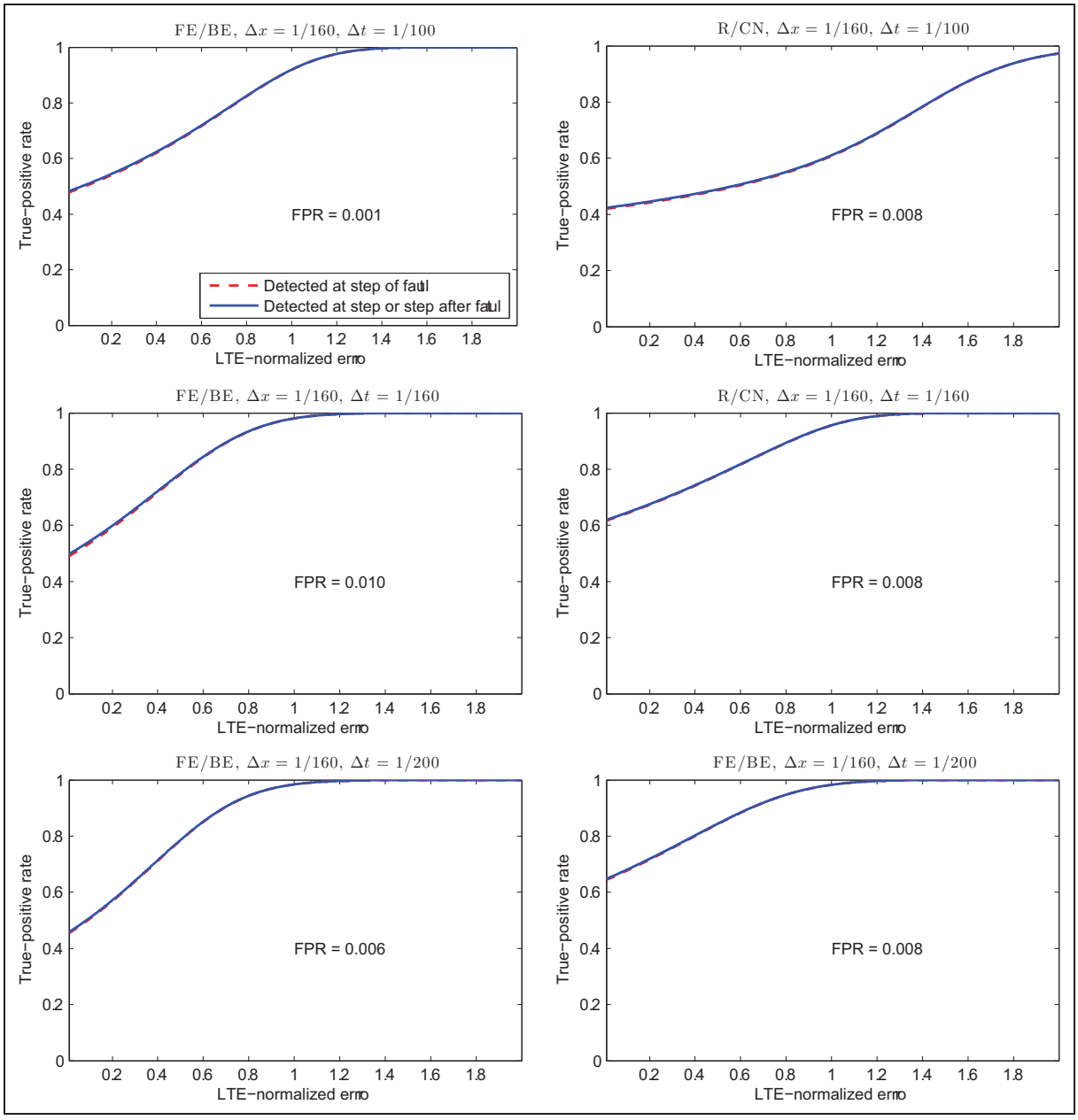

Figure 7. Detection results for heat equation under Configuration 3 with $dt = 1/100$, $1/160$, and $1/200$. Faults are injected by multiplying a single component of the solution at the previous time step by a normally distributed random variable with mean 1 and variance 1×10^{-6} (R/CN) or 1×10^{-4} (FE/BE).

detected. However, there are some anomalies that are so well 'disguised' that they are indistinguishable from normal iterations, and only by allowing an extreme increase in FPR are we able to detect these.

5.4 Detection indicators for the heat equation

In equations (5) and (6), we defined the indicators J_n and V_n used by the detector's two-indicator strategy. J_n measured the jump in the differences in the sequence, and V_n measured the change in variance of the differences. We call these the *relative jump* and the *variance change*.

We now empirically explore the advantages of the two-indicator strategy over a single detection indicator. Figure 8 shows the detection results for FE/BE when using only the relative jump and only the variance change under Configuration 2 of the heat equation with $\Delta t = 1/200$. We used the same injected faults as in Section 5.3. In other words, Figure 8 shows the performance of the individual detectors when compared to the two-indicator strategy in Figure 6.

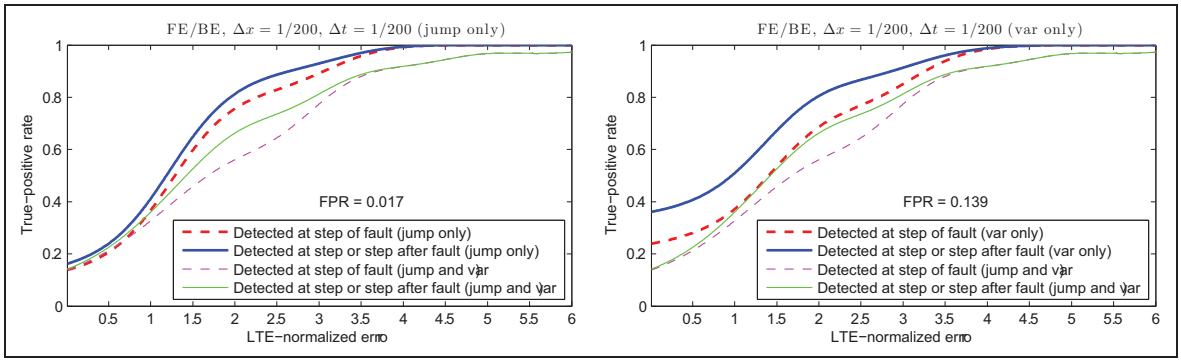

Figure 8. Detection results for FE/BE on the heat equation under Configuration 2 with $\Delta t = 1/200$. The left and right plots compare using only the relative jump or only the variance change, respectively, to using both relative jump and variance change. The listed FPR is for using only the relative jump (left) or only the variance (right). When using both indicators, there are no false positives. Faults were injected by multiplying a single component of the solution at the previous time step by a normally distributed random variable with mean 1 and variance 5×10^{-5}. Identical faults were injected for each type of detector.

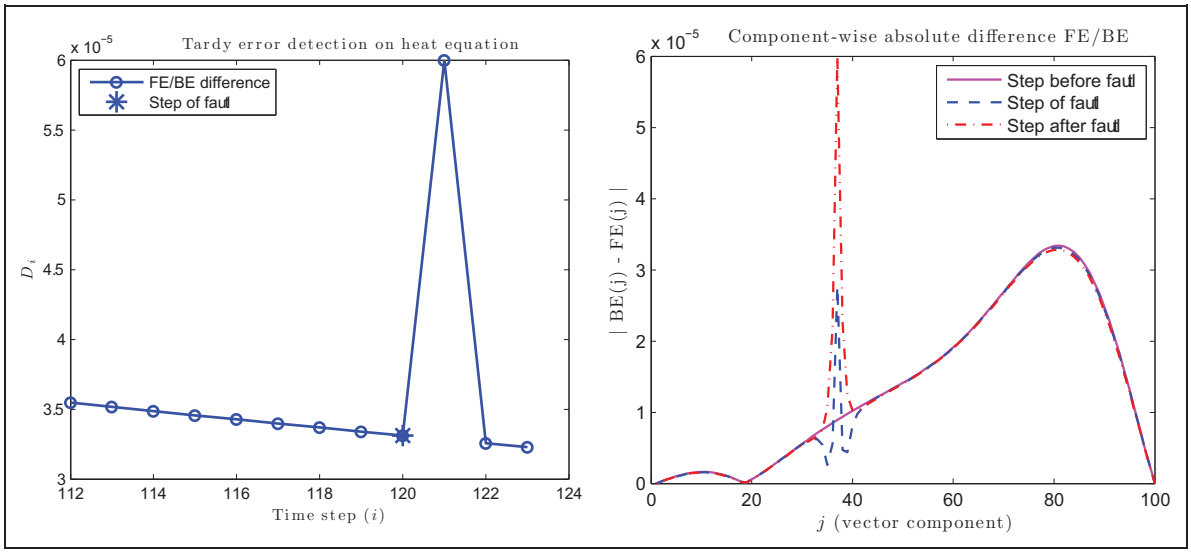

Figure 9. The left plot shows the infinity-norm difference between forward and backward Euler schemes under Configuration 1 of the heat equation with $\Delta t = 1/100$. At the step of the fault, there is a small jump in the difference, while at the step after the fault, there is a large spike. The right plot shows the component-wise absolute difference of forward and backward Euler solutions at the time steps before, during which, and after the fault occurs. The fault corrupts the source term in the 36th component, and a clear spike is seen at that location.

When the two-indicator strategy was used (Figure 6), no false positives were produced. The FPR increases to 14% when using only the relative jump and to 2% when using only the variance change. The TPR is only mildly increased when using only the relative jump and is mostly unchanged when using only the variance change. This suggests that the FPR can be dramatically reduced by employing the two-indicator strategy.

5.5 Tardy error detection with the heat equation

In some configurations of the heat equation, it is common to detect the error one time step *after* the fault occurs (see, for example, Figure 5). The left plot of Figure 9 shows D_i for the FE/BE $\mathcal{A}/\mathcal{B}$ formulation near the point of the injected fault for one of the simulations in which tardy error detection occurs. The right plot of Figure 9 shows the component-wise difference of the solution vectors near the time step of the fault (recall that D_i is the infinity-norm difference). We see no jump in D_i at the step of the fault and a large spike the step after the fault. Some of the components of the difference between backward and forward Euler are naturally larger than others, and the fault occurs at a spatial location where the differences tend to be small.

Why does the jump occur the step after the fault. In this case, we are corrupting the source term q in

equation (4). At the fault, only backward Euler uses the corrupted evaluation of q (forward Euler uses the value of q from the *previous* time step). The implicit nature of the backward Euler scheme forces the new solution to 'agree' with the corrupted source term, and the result is a small change in the difference of the solution vector. At the step after the fault, forward Euler uses the corrupted source term. Since forward Euler is explicit, the corrupted value is 'accepted' and taken for a full time step. This causes the large spike in the sequence of D_i to occur at the step *after* the fault.

The right plot of Figure 9 illustrates the advantages and disadvantages of using the infinity-norm. On the one hand, the errors tend to be localized spatially, and local spikes are easier to detect with the infinity-norm. However, when the solution vector difference has different scales, it is more difficult to detect faults that occur in spatial locations where the solution vector difference is smaller. In general, we found that the infinity-norm worked better than the 1-norm and 2-norm. We note that our general framework does restrict D_i to considering only a single norm. However, for simplicity, we chose a single norm for our experiments. These results show that examining D_i locally in space and in time can be beneficial, and this is an area of future work.

5.6 Incompressible Navier–Stokes equations

The incompressible Navier–Stokes equations in two dimensions with no external forces are

$$u_t = -(u^2)_x - (uv)_y + \tfrac{1}{Re}\nabla\cdot\nabla u - p_x$$
$$v_t = -(v^2)_y - (uv)_x + \tfrac{1}{Re}\nabla\cdot\nabla v - p_y$$

Here, u and v are the velocity components, Re is the Reynolds number, and p is the pressure.

In our experiments, $\mathcal{B}$ is based on a simple projection method in Strang (2007, Section 6.7) and open-source Matlab code (Seibold, 2008). The boundary is a square, and the boundary conditions are those of a driven cavity flow.

Let $U_{\mathcal{B}}^n$, $V_{\mathcal{B}}^n$, and $P_{\mathcal{B}}^n$ be the numerical solutions at the nth time step and let Δt be the time step. The overall structure of the update to $U_{\mathcal{B}}$ in an iteration of $\mathcal{B}$ is as follows:

1. Explicit (forward-Euler-like) handling of nonlinear terms:

$$\frac{U_{\mathcal{B}}^* - U_{\mathcal{B}}^n}{\Delta t} = -((U_{\mathcal{B}}^n)^2)_x - (U_{\mathcal{B}}^n V_{\mathcal{B}}^n)_y$$

where the subscripts denote centered difference.

2. Implicit solve for viscous term:

$$\frac{U_{\mathcal{B}}^{**} - U_{\mathcal{B}}^*}{\Delta t} = \frac{1}{Re}\nabla\cdot\nabla U_{\mathcal{B}}^{**} \tag{7}$$

3. Solve for the pressure correction:

$$\nabla\cdot\nabla P_{\mathcal{B}}^{n+1} = \frac{1}{\Delta t}\left((U_{\mathcal{B}}^{**})_x + (V_{\mathcal{B}}^{**})_y\right) \tag{8}$$

4. Update the solution:

$$U_{\mathcal{B}}^{n+1} = U_{\mathcal{B}}^{**} - \Delta t(P_{\mathcal{B}}^{n+1})_x \tag{9}$$

The update to $V_{\mathcal{B}}$ follows analogous steps.

For our experiments, we use extrapolation for the auxiliary scheme,

$$U_{\mathcal{A}}^{n+1} = 2U_B^n - U_B^{n-1}, \quad V_{\mathcal{A}}^{n+1} = 2V_B^n - V_B^{n-1}$$

along with the difference computation

$$D_n = \max\left(\max_{i,j}|(U_{\mathcal{B}}^n)_{ij} - (U_{\mathcal{A}}^n)_{ij}|, \max_{i,j}|(V_{\mathcal{B}}^n)_{ij} - (V_{\mathcal{A}}^n)_{ij}|\right)$$

There is no requirement for the $\mathcal{A}/\mathcal{B}$ scheme to encompass the entire numerical method. We could implement specialized $\mathcal{A}/\mathcal{B}$ schemes for each of the three steps in addition to or in place of the extrapolation scheme. An advantage of compartmentalized $\mathcal{A}/\mathcal{B}$ schemes is that we can detect an error early in the iteration and avoid doing extra computation. However, extrapolation is simple and demonstrates that detecting silent errors in a nonlinear PDE system need not involve a lot of extra work.

Our experiments use the above projection method on the spatial domain $[0,1]\times[0,1]$ for $t\in[0,2]$. The discretizations are $\Delta x = \Delta y = 1/40$ and $\Delta t = 1/100$. We performed two simulations with different Reynolds numbers and different types of data corruption. In the first simulation, $Re = 2000$, and we corrupted the previous solution, $U_{\mathcal{B}}^n$ ($\sigma^2 = 5\times10^{-1}$). In the second simulation, $Re = 20$, and we corrupted the right-hand side of the linear system in equation (8) ($\sigma^2 = 2$). Each simulation consisted of 2000 trials. In each trial, the corruption occurred at a single time step, chosen uniformly at random. The entry in $U_{\mathcal{B}}^n$ and the entry on the right-hand side of equation (8) were chosen uniformly at random.

The TPR as a function of the LTE-normalized error is in Figure 10. The results are similar to the behavior of the detector for the Van der Pol equation and the heat equation. Errors with a large LTE-normalized error are easily detected, and the FPR is small.

6. Discussion

We proposed a general method for detecting silent errors in time-stepping schemes. The central idea is to use a cheap checking method to compare against the primary numerical algorithm. In Section 2, we describe the general approach, which is applicable to iterative computations, and the ideas in Sections 3 and 5 can be

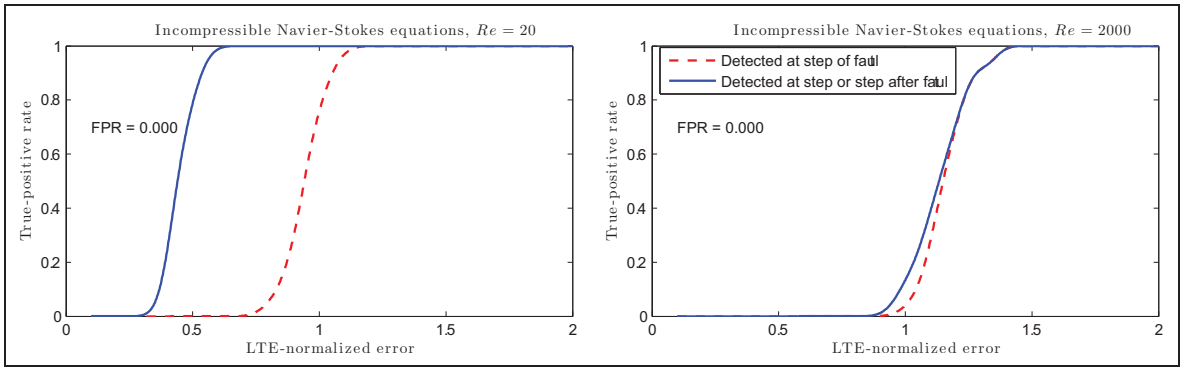

Figure 10. Detector performance on incompressible Navier–Stokes equations with $Re = 2000$ (left) and $Re = 20$ (right). In the left plot, errors are introduced by multiplying a previous entry of the numerical solution of one velocity component (U) by a normal random variable with mean 1 and variance 5×10^{-1}. In the right plot, errors are introduced by multiplying an entry on the right-hand side of the linear system in equation (8) by a normal random variable with mean 1 and variance 2.

extended to algorithms not discussed in this paper. For example, extrapolation (Section 3.3) is a simple checking scheme that is available for nearly all iterative algorithms. Although the scope of this paper is limited to iterative computations, a checking scheme is applicable for nearly all numerical algorithms. Finding efficient checking schemes for a broader class of numerical algorithms is an area of future work.

By comparing the results of a base time-stepping scheme and an auxiliary scheme, we are able to detect almost all significant errors. The auxiliary scheme is readily available for standard ODE solvers such as Runge–Kutta and LMMs, as well as for PDE solvers for the heat equation and the Navier–Stokes equations. In several simulations, our detection scheme successfully flags nearly all LTE-normalized errors above five, while maintaining an overall FPR of less than 10% (and in many cases, 0%). An important property of our detection scheme is that it is most successful detecting errors that have the largest impact on the solution. We measure the impact by the LTE-normalized error.

Our method requires additional implementation by the application developer. For example, when using the FE/BE $\mathcal{A}/\mathcal{B}$ method, we cannot simply call a forward Euler solver. The forward Euler method needs to use the base method's solution from the previous time step. Furthermore, the developer needs to select the appropriate checking scheme. However, the computational kernels remain the same, so the developer does not need to re-write entire applications from scratch. With mature, modular software packages such as Trilinos (Heroux et al., 2005) and PETSc (Balay et al., 2013), we hope that implementation will not be a major obstacle.

One area of future work is a more formal analysis of the errors in the difference schemes and the sequence D_i. We would like to say that a fault *must* have occurred if some D_i was above a computed threshold. Typical error bounds are too loose with constants to be practical, so careful analysis is needed.

Further characterizations of silent errors is another area of future work. First, it would be useful to detect *what* caused the fault. We can use data checksums to determine whether a previous solution was corrupted, but determining if a function evaluation caused an error is more difficult.

Second, we would like to know *where* the fault occurred. For example, when perturbing an entry in the source term of the heat equation (the function q), the heat is dissipated locally near the spatial point of perturbation. The solution vector is then perturbed near (in space) to where the source term was perturbed. We saw this phenomenon in Figure 9, and it led to tardy error detection, which we discussed in Section 5.5. Thus, it is possible to detect where (physically) the fault occurred. This is important for two reasons. First, we can improve the performance of our error detector if we look for errors in space *and* time. Second, in parallel solvers, it is common for different spatial locations to be assigned to different processors. By detecting the point in space where the fault occurred, we have an idea of which processor experienced a silent error. In other physical simulations where perturbations cause local changes, we can apply the same idea.

The spatial location of the error is also potentially important in improving the sensitivity of the detector. As shown in Figure 9, it is possible that detectors that are local in both space and time may be a profitable extension of the approach taken here.

Funding

We thank the US Department of Energy, which supported this work (award number DE - SC0005026). Austin R Benson is also supported by an Office of Technology Licensing Stanford Graduate Fellowship. Sven Schmit is also supported by the Prins Bernhard Cultuurfonds.

References

Balay S, Brown J, Buschelman K, et al. (2013) PETSc users manual, revision 3.4. Available at: http://www.mcs.anl.gov/petsc/petsc-current/docs/manual.pdf (accessed 24 March 2014).

Berger MJ and Oliger J (1984) Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics* 53(3): 484–512.

Bronevetsky G and de Supinski B (2008) Soft error vulnerability of iterative linear algebra methods. In: *Proceedings of the 22nd annual international conference on supercomputing*, New York, NY, pp. 155–164.

Cappello F, Geist A, Gropp B, et al. (2009) Toward exascale resilience. *International Journal of High Performance Computing Applications* 23(4): 374–388.

Casas M, de Supinski BR, Bronevetsky G, et al. (2012) Fault resilience of the algebraic multi-grid solver. In: *Proceedings of the 26th ACM international conference on supercomputing*, New York, NY, pp. 91–100.

Dongarra J, Beckman P, Moore T, et al. (2011) The international exascale software project roadmap. *International Journal of High Performance Computing Applications* 25(1): 3–60.

Dormand JR and Prince PJ (1980) A family of embedded Runge-Kutta formulae. *Journal of Computational and Applied Mathematics* 6(1): 19–26.

Du P, Luszczek P and Dongarra J (2012) High performance dense linear system solver with resilience to multiple soft errors. In: *International conference on computational science, ICCS 2012*, Omaha, NE.

Fehlberg E (1969) Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems. Technical report R-315, National Aeronautics and Space Administration. Available at: http://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19690021375.pdf

Fujita H, Schreiber R and Chien AA (2013) It's time for new programming models for unreliable hardware (extended abstract). In: *Provocative ideas session, ASPLOS*.

Hamilton JD (1994) *Time Series Analysis*, vol. 2. Cambridge: Cambridge University Press.

Heroux MA, Bartlett RA, Howle VE, et al. (2005) An overview of the Trilinos Project. *ACM Transactions on Mathematical Software (TOMS)* 31(3): 397–423.

Hoemmen M and Heroux MA (2011) Fault-tolerant iterative methods via selective reliability. Technical report SAND2011-3915 C, Sandia National Laboratories.

Huang KH and Abraham JA (1984) Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers* 100(6): 518–528.

Lin J, Keogh E, Lonardi S, et al. (2003) A symbolic representation of time series, with implications for streaming algorithms. In: *Proceedings of the 8th ACM SIGMOD workshop on research issues in data mining and knowledge discovery*, pp. 2–11.

Nightingale EB, Douceur JR and Orgovan V (2011) Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In: *Proceedings of the sixth conference on computer systems*, New York, NY, pp. 343–356.

Rinard M (2013) Parallel synchronization-free approximate data structure construction. In: *Proceedings of the 5th USENIX workshop on hot topics in parallelism*. Available at: https://www.usenix.org/conference/hotpar13/parallel-synchronization-free-approximate-data-structure-construction (accessed 1 August 2013).

Seibold B (2008) A compact and fast Matlab code solving the incompressible Navier-Stokes equations on rectangular domains. Available at: http://math.mit.edu/cse/codes/mit18086_navierstokes.pdf. (accessed 15 January 2014)

Shi G, Enos J, Showerman M, et al. (2009) On testing GPU memory for hard and soft errors. In: *Proceedings of the symposium on application accelerators in high-performance computing*.

Snir M, Wisniewski RW, Abraham JA, et al. (2013) Addressing failures in exascale computing. *International Journal of High Performance Computing Applications*. Epub ahead of Print 21 March 2014. DOI: 10.1177/1094342014522573.

Strang G (2007) *Computational Science and Engineering*. Wellesley, MA: Wellesley-Cambridge Press.

Strikwerda J (2007) *Finite Difference Schemes and Partial Differential Equations*. Philadelphia, PA: SIAM.

Van Dam HJJ, Vishnu A and de Jong WA (2013) A case for soft error detection and correction in computational chemistry. *Journal of Chemical Theory and Computation* 9(9): 3995–4005.

Zheng G, Ni X and Kalé LV (2012) A scalable double in-memory checkpoint and restart scheme towards exascale. In: *2012 IEEE/IFIP 42nd international conference on dependable systems and networks workshops (DSN-W)*, pp. 1–6.

Author biographies

Austin R Benson is a graduate student at Stanford's Institute for Computational and Mathematical Engineering and an Office of Technology Licensing Stanford Graduate Fellow. His research includes parallel algorithms for scientific computing and algorithms for structured matrices. In the summer and fall of 2013, he was a research intern at HP Labs, working with Robert Schreiber. Previously, Benson obtained a BS in Computer Sciences and Engineering and a BA in Applied Mathematics from the University of California, Berkeley.

Sven Schmit is a graduate student at the Institute for Computational and Mathematical Engineering at Stanford University. His research interests span topics in statistics, computer science, and applied mathematics. In the summer of 2013, he interned at HP Labs, working under supervision of Robert Schreiber. Schmit holds a BSc in Econometrics and Operations Research from the University of Groningen, and a MASt in Mathematics from the University of Cambridge.

Robert Schreiber is a Distinguished Technologist at Hewlett Packard Laboratories. Schreiber's research

spans sequential and parallel algorithms for matrix computation, compiler optimization for parallel languages, and high performance computer design. With Cleve Moler and John Gilbert, he developed the sparse matrix extension of Matlab. He created the NAS CG parallel benchmark. He was a designer of the High Performance Fortran language. At HP, Schreiber led the development of PICO, a system for the synthesis of custom hardware accelerators. His recent work concerns architectural uses of CMOS nanophotonic communication and nonvolatile memory architecture. He is an ACM Fellow, a SIAM Fellow, and was awarded, in 2012, the Career Prize from the SIAM Activity Group in Supercomputing.