Masterproef scriptie

Recursive Monte Carlo for linear ODEs

Auteur: *Isidoor Pinillo Esquivel*

Promotor: *Wim Vanroose*

Academiejaar 2022-2023

Contents

Abstract

This thesis explores applying recursive Monte Carlo for solving linear ordinary differential equations with a vision towards partial differential equations. The proposed algorithms capitalize on the appropriate combination of Monte Carlo techniques. These Monte Carlo techniques get introduced with examples and code.

1 Introduction

1.1 Notation

Notations utilized include:

- $B(p) \sim \text{Bernoulli}(p) = \begin{cases} 1 & \text{if } U < p \\ 0 & \text{else} \end{cases}$
- BVP = Boundary Value Problem
- $H(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{else} \end{cases}$
- i.i.d. = independent identically distributed
- $U \sim \text{Uniform}(0, 1)$
- IBC = Information-Based Complexity
- IVP = Initial Value Problem
- MC = Monte Carlo
- ODE = Ordinary Differential Equation
- PDE = Partial Differential Equation
- RMC = Recursive Monte Carlo
- RRMC = Recursion in Recursion Monte Carlo
- RRVE (Recursive Random Variable Equation): An equation that defines a family of random variables in terms of itself
- RV = Random Variable
- Random variables will be denoted with capital letters, e.g., X, Y or Z.
- WoS = Walk-on-Spheres

1.2 Related Work

The primary motivating paper for this work is the work by Sawhney et al. (2022) [Saw+22], which introduces the Walk-on-Sphere (WoS) method for solving second-order elliptic PDEs with varying coefficients and Dirichlet boundary conditions. Their techniques have shown high accuracy even in the presence of geometrically complex boundary conditions. We were inspired to apply the underlying mechanics of these Monte Carlo (MC) techniques to ODEs to explore parallel in time and the possibility to extend their techniques to other types of PDEs.

We made an interactive data map of the literature read mainly in function of this thesis available at `https://huggingface.co/spaces/ISIPINK/zotero_map`. It may take 10 seconds to load.

The latest paper that we found on an unbiased IVP solver is by Ermakov and Smilovitskiy's 2021 [ES21] that study an unbiased method for a Cauchy problem for large systems of linear ODEs for describing queuing systems. Similarly to us they base their solver on Volterra integral equations.

Other literature is a bit further away. The most important fields we draw from are:

- rendering and WoS/first passage literature which contain many practical recursive MC techniques,
- stochastic gradient descent literature that is connected through continuous gradient descent which we first encountered in [Hua+17],
- Information-Based Complexity (IBC) literature which was unexpected to us, there are some intrestring biased algorithms applied on ODEs that achieve optimal IBC rates for RMSE for some smoothness classes, similar to us Daun's 2011 [Dau11] uses control variates to achieve optimal IBC.

A recurrent theme in these fields is that optimal IBC algorithms and unbiased algorithms are of theoretical importance.

1.3 Contributions

TODO: add intuition and motivations to the beginning of sections and do section descriptions here.

A significant part of this thesis is dedicated to informally introducing Recursive Monte Carlo (RMC) and applying variance reduction techniques for ODEs.

The key contribution is an unbiased MC method (3.3.3) for linear IVPs for systems of ODEs by using recursion in recursion and variance reduction techniques. We conjecture that this method achieves optimal IBC.

2 Background

2.1 Monte Carlo Integration

In this subsection, we review basic MC theory.

MC integration is any way to use random sampling to estimate an integral.

Definition 2.1.1 (Uniform Monte Carlo Integration)
We define uniform Monte Carlo integration of $f : \mathbb{R}^n \to \mathbb{R}^m$ over $\Omega \subset \mathbb{R}^n$ as an estimation of the average of $f(U)$, with $U \sim \text{Uniform}(\Omega)$. Combined with the Best Linear Unbiased Estimators (BLUEs) for the average, MC Integration in that case can be summarized in the following formula:

$$\int_\Omega f(s)ds \approx \frac{1}{n}\sum_{i=1}^{n} f(U_j), \tag{1}$$

where n is the amount of samples used and U_j i.i.d. U.

Because estimators are Random Variables (RV) the cost and error also are RVs. In most cases these RVs are hard to impossible to get. Directly making comparisons between estimators based on these RVs is hard, there is no Pareto front we can draw. Instead statistics of these variables are used to do comparisons.

Accuracy comparisons between estimators are usually done with (root-)mean-square error (RMSE).

Definition 2.1.2 (Root-Mean-Square Error)
We define the Root-Mean-Square Error (RMSE) of an estimator $\tilde{\theta}$ for θ as follows:

$$\text{RMSE}(\tilde{\theta}) = \sqrt{E[||\tilde{\theta} - \theta||_2^2]}. \tag{2}$$

Even comparisons based on RMSE can be counterintuitive look at Stein's paradox for example. We will always limit us to 1 dimensional unbiased estimators, so that MSE becomes the variance. Estimating variance is simple and can be used to get confidence intervals via Chebyshev's inequality or an approximate normal distribution argument.

Average (FLOPs or time)/simulation are very common cost statistics. We can imagine (memory or wall time) at risk (similar to value at risk) to be useful to keep track of. The statistics are problem dependent.
If we limit ourselves to equation (2) with big sample size and finite variance assumption on error and simulation time. Memory needs to be just enough to evaluate f and to keep an accumulation of the sum which is negligible. Computation across simulations is embarrassingly parallel which make them well suited to a GPU implementation. These assumptions get you pretty close to optimal solutions and are very practical for high dimensions or complex domains. With these assumption and a linear trade off between average simulation time and variance allows us to define a relative efficiency statistic for comparing estimators.

Definition 2.1.3 (Relative Monte Carlo Efficiency)
Define relative Monte Carlo efficiency of an estimator F as follows:

$$\epsilon[F] = \frac{1}{\text{Var}[F]T[F]}, \tag{3}$$

with T the average simulation time.

Related Work 2.1.4 (Relative Monte Carlo Efficiency)
For a reference see [Vea97] page 45.

For smooth 1 dimensional integration the linear trade off between variance and average simulation time is not even close to optimal see theorem 2.4.5.
Comparing better trades offs is usually done with Information-Based Complexity (IBC). IBC is more of a qualitative measure and doesn't directly imply that an algorithm is practical. We won't define IBC rigorously.

Definition 2.1.5 (Information-Based Complexity)
IBC is a way to describe asymptotically (for increasing accuracy/function calls) the trade-off between the average amount of function calls (information) needed and accuracy.

Example 2.1.6 (IBC of equation (2))
In equation (2) the function calls trades of linearly with variance. For n function calls the RMSE $= O\left(\frac{1}{\sqrt{n}}\right)$ or equivalently if we want a RMSE of ε we would need $O\left(\frac{1}{\varepsilon^2}\right)$ function calls.

2.2 Recursive Monte Carlo

In this subsection, we introduce recursive Monte Carlo with the following initial value problem:

$$y' = y, \quad y(0) = 1. \tag{4}$$

By integrating both sides of equation (4), we obtain:

$$y(t) = 1 + \int_0^t y(s)ds. \tag{5}$$

Equation (5) represents a recursive integral equation, specifically a linear Volterra integral equation of the second type. By estimating the recursive integral in equation (5) using Monte Carlo, we derive the following estimator:

$$Y(t) = 1 + ty(Ut), \tag{6}$$

where $U \sim \text{Uniform}(0, 1)$. If y is well-behaved, then $E[Y(t)] = y(t)$, but we cannot directly simulate $Y(t)$ without access to $y(s)$ for $s < t$. However, we can replace y with an unbiased estimator without affecting $E[Y(t)] = y(t)$, by the law of total expectation ($E[X] = E[E[X|Z]]$). By replacing y with Y itself, we obtain a recursive expression for Y:

$$Y(t) = 1 + tY(Ut). \tag{7}$$

Equation (7) is a recursive random variable equation. If one were to try to simulate Y with equation (7), it would recurse indefinitely. To overcome this, approximate $Y(t) \approx 1$ near $t = 0$ introducing minimal bias. Later, we will discuss Russian roulette (2.3.2), which can be used as an unbiased stopping mechanism.

Python Code 2.2.1 (implementation of (7))

```
from random import random as U
def Y(t, eps): return 1 + t*Y(U()*t, eps) if t > eps else 1
def y(t, eps, nsim):
    return sum(Y(t, eps) for _ in range(nsim))/nsim
print(f"y(1) approx {y(1,0.01,10**3)}")
# y(1) approx 2.710602603240193
```

To gain insight into realizations of recursive random variable equations, it can be helpful to plot all recursive calls $(t, Y(t))$, as shown in Figure 1 for this implementation.

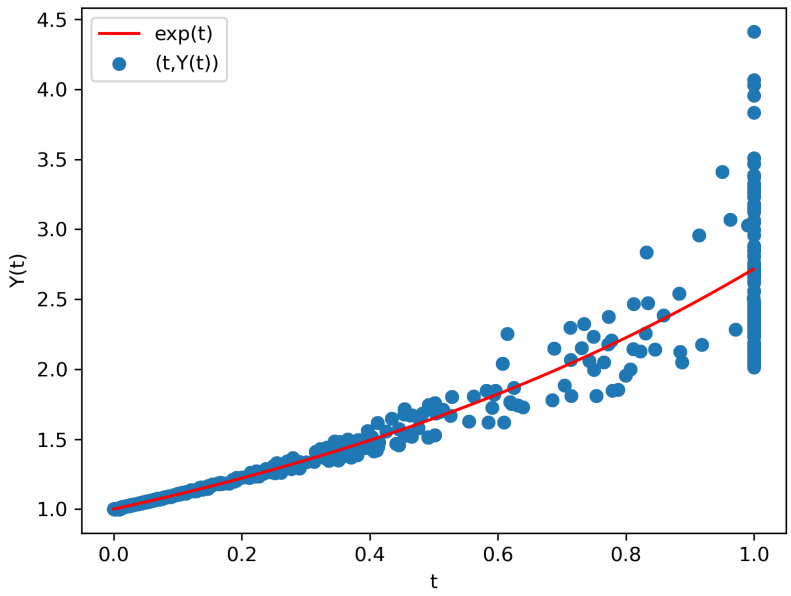

Figure 1: Recursive calls of (2.2.1)

Notice that the variance of (2.2.1) increases rapidly as t increases. This issue gets resolved later (see (3.3.1)).

2.3 Modifying Monte Carlo

In this subsection, we discuss techniques for modifying RRVEs in a way that preserves the expected value of the solution while acquiring more desirable properties. These techniques are only effective when applied in a smart way by using prior information about the problem or information about costs.

We will be frequently interchanging RVs with the same expectance. That is why we introduce the following notation.

Notation 2.3.1 ($\cong$)

$$X \cong Y \iff E[X] = E[Y].$$

Russian roulette is a MC technique commonly employed in rendering algorithms. The concept behind Russian roulette is to replace a RV with a less computationally expensive approximation sometimes.

Definition 2.3.2 (Russian roulette)

We define Russian roulette on X with free parameters Y_1, Y_2 ($E[Y_1] = E[Y_2]$), $p \in [0, 1]$ and U independent of Y_1, Y_2, X as follows:

$$X \cong \begin{cases} \frac{1}{p}(X - (1-p)Y_1) & \text{if } U < p \\ Y_2 & \text{else} \end{cases}. \tag{8}$$

Example 2.3.3 (Russian roulette)

Let us consider the estimation of $E[Z]$, where Z is defined as follows:

$$Z = U + \frac{f(U)}{1000}. \tag{9}$$

Here, $f : \mathbb{R} \to [0, 1]$ is an expensive function to compute. Directly estimating $E[Z]$ would involve evaluating f in each simulation, which can be computationally costly. To address this, we can modify Z to:

$$Z \cong U + B\left(\frac{1}{100}\right)\frac{f(U)}{10}. \tag{10}$$

This RV requires calling f on average once every 100 simulations. This significantly reduces the computational burden while increasing the variance slightly thereby increasing the MC efficiency.

In this case it is also possible to estimate the expectance of the 2 terms of Z separately. Given the variances and computational complexity of both terms, you can calculate the asymptotic optimal division of samples for each term. But this isn't the case anymore with RMC.

Example 2.3.4 (Russian roulette on (7))

To address the issue of indefinite recursion in equation (7), Russian roulette can be employed by approximating the value of Y near $t = 0$ with 1 sometimes. Specifically, we replace the coefficient t in front of the recursive term with $B(t)$ when $t < 1$. The modified recursive expression for $Y(t)$ becomes:

$$y(t) \cong Y(t) = \begin{cases} 1 + B(t)Y(Ut) & \text{if } t < 1 \\ 1 + tY(Ut) & \text{else} \end{cases}. \tag{11}$$

Python Code 2.3.5 (implementation of (11))

```python
from random import random as U
def Y(t):
    if t>1: return 1 + t*Y(U()*t)
    return 1 + Y(U()*t) if U() < t else 1
def y(t, nsim): return sum(Y(t) for _ in range(nsim))/nsim
print(f"y(1) approx {y(1,10**3)}")
# y(1) approx 2.698
```

Interestingly, $\forall t \leq 1 : Y(t)$ is constrained to take on only integer values. This is visually clear in Figure 2.

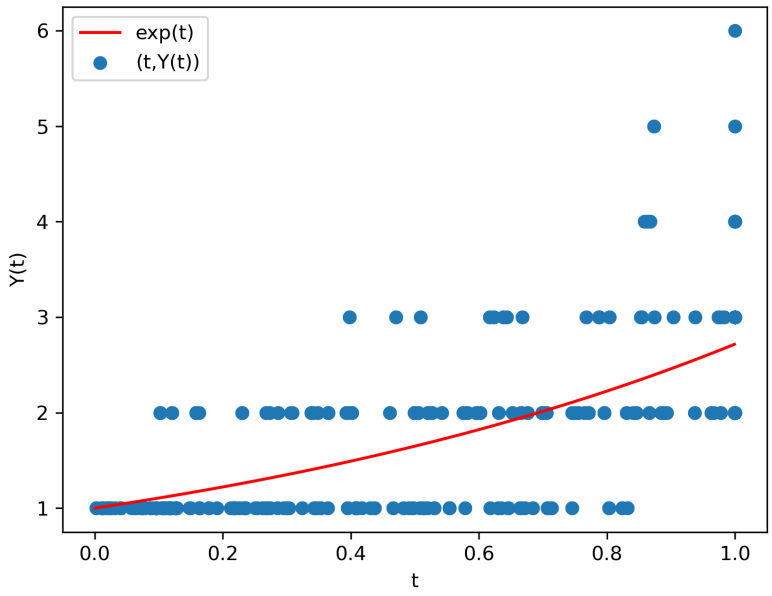

Figure 2: Recursive calls $(t, Y(t))$ of (2.3.5)

Splitting is a technique that has almost the reverse effect of Russian roulette. Instead of reducing the number of simulations of a RV as Russian roulette does, we increase it by using more samples (i.e., splitting the sample) which reduces the variance.

Definition 2.3.6 (splitting)
Splitting X refers to utilizing multiple $X_j \sim X$ (not necessarily independent) to reduce variance by taking their average:

$$X \cong \frac{1}{N} \sum_{j=1}^{N} X_j. \tag{12}$$

Splitting the recursive term in a RRVE can result in additive branching recursion, necessitating cautious management of terminating the branches promptly to prevent exponential growth in computational complexity. To accomplish this, termination strategies that have been previously discussed can be employed. Subsequently, we will explore the utilization of coupled recursion as a technique to mitigate additive branching recursion in RRVEs (refer to (3.2.3)).

Example 2.3.7 (splitting on (7))
We can "split" the recursive term of Equation (7) into two parts as follows:

$$y(t) \cong Y(t) = 1 + \frac{t}{2}(Y_1(Ut) + Y_2(Ut)), \tag{13}$$

where $Y_1(t)$ and $Y_2(t)$ are i.i.d. with $Y(t)$.

Python Code 2.3.8 (implementation of (2.3.7))

```
from random import random as U
def Y(t):
    u = U()
    if t > 1: return 1 + t*(Y(u*t)+Y(u*t))/2
    return 1 + (Y(u*t)+Y(u*t))/2 if U() < t else 1
def y(t, nsim): return sum(Y(t) for _ in range(nsim))/nsim
print(f"y(1) approx {y(1,10**3)}")
# y(1) approx 2.73747265625
```

Definition 2.3.9 (control variates)
Define control variating $f(X)$ with $\tilde{f}$ an approximation of f as:

$$f(X) \cong (f - \tilde{f})(X) + E[\tilde{f}(X)]. \tag{14}$$

Note that control variating requires the evaluation of $E[\tilde{f}(X)]$. When this is estimated instead of evaluated exactly, we refer to it as 2-level MC.

Example 2.3.10 (control variate on (7))
To create a control variate for Equation (7) that effectively reduces variance, we employ the approximation $y(t) \approx \tilde{y} = 1 + t$ and define the modified recursive term as follows:

$$Y(t) = 1 + E[\tilde{y}(Ut)] + t(Y(Ut) - \tilde{y}(Ut)) \tag{15}$$

$$= 1 + t + \frac{t^2}{2} + t(Y(Ut) - 1 - Ut). \tag{16}$$

It is important to note that while we could cancel out the constant term of the control variate, doing so would have a negative impact on the Russian roulette implemented later.

Python Code 2.3.11 (implementation of (2.3.10))

```
from random import random as U
def Y(t):
    u = U()
    if t > 1: return 1+t**2/2 + t*(Y(u*t)-u*t)
    return 1 + t + t**2/2 + (Y(u*t)-1-u*t if U() < t else 0)
def y(t, nsim): return sum(Y(t) for _ in range(nsim))/nsim
print(f"y(1) approx {y(1,10**3)}")
# y(1) approx 2.734827303480301
```

Related Work 2.3.12 (MC modification)
For further reference on these techniques see [Vea97].

2.4 Monte Carlo Trapezoidal Rule

In this subsection, we introduce a MC trapezoidal rule that exhibits similar convergence behavior to the methods discussed later. The MC trapezoidal rule is essentially a regular Monte Carlo method enhanced with control variates based on the trapezoidal rule.

Definition 2.4.1 (MC trapezoidal rule)
We define the MC trapezoidal rule for approximating the integral of function f over the interval $[x, x+\Delta x]$ with a Russian roulette rate l and $\tilde{f}$ the linear approximation of f corresponding to the trapezoidal rule as follows:

$$\int_x^{x+\Delta x} f(s)ds \tag{17}$$

$$= \int_x^{x+\Delta x} \tilde{f}(s)ds + \int_x^{x+\Delta x} f(s) - \tilde{f}(s)ds \tag{18}$$

$$= \Delta x \frac{f(x) + f(x+\Delta x)}{2} + E\left[f(S) - \tilde{f}(S)\right] \tag{19}$$

$$\cong \Delta x \frac{f(x) + f(x+\Delta x)}{2} + \Delta x l B\left(\frac{1}{l}\right)\left(f(S) - f(x) - \frac{S-x}{\Delta x}\left(f(x+\Delta x) - f(x)\right)\right), \tag{20}$$

where $S \sim \text{Uniform}(x, x+\Delta x)$.

Lemma 2.4.2 (RMSE MC Trapezoidal Rule)
The MC trapezoidal rule for a twice differentiable function has

$$\text{RMSE} = O\left(\Delta x^3\right). \tag{21}$$

Proof. Start from equation (20). The MSE is the variance so we can ignore addition by constants.

$$\text{MSE} = \text{Var}\left(\Delta x l B\left(\frac{1}{l}\right)\left(f(S) - f(x) - \frac{S-x}{\Delta x}\left(f(x+\Delta x) - f(x)\right)\right)\right) \tag{22}$$

We substitute $S = \Delta x U + x$ and then apply Taylor's theorem finishing the proof:

$$\text{MSE} = \text{Var}\left(\Delta x l B\left(\frac{1}{l}\right)\left(f(xU+x) - f(x) - U\left(f(x+\Delta x) - f(x)\right)\right)\right) \tag{23}$$

$$= \text{Var}\left(\Delta x l B\left(\frac{1}{l}\right)\left(U\Delta x f'(x) + \frac{U^2\Delta x^2}{2} f''(Z_1) - U\left(\Delta x f'(x) + \Delta x^2 f''(z_2)\right)\right)\right) \tag{24}$$

$$= \text{Var}\left(\Delta x l B\left(\frac{1}{l}\right)\left(\frac{U^2\Delta x^2}{2} f''(Z_1) - \frac{U\Delta x^2}{2} f''(z_2)\right)\right) \tag{25}$$

$$= \Delta x^6 \text{Var}\left(l B\left(\frac{1}{l}\right)\left(\frac{U^2}{2} f''(Z_1) - \frac{U}{2} f''(z_2)\right)\right), \tag{26}$$

for some $Z_1 \in [x, S], z_2 \in [x, x+\Delta x]$. □

Note that the proof doesn't rely on Russian roulette ($l = 1$).

Definition 2.4.3 (composite MC trapezoidal rule)
Define the composite MC trapezoidal rule for approximating the integral of function f over the interval $[a, b]$ for a uniform grid (x_j) with n intervals and a Russian roulette rate l as follows:

$$\int_a^b f(s)ds \cong \Delta x \sum_{j=1}^{n} \frac{f(x_j) + f(x_j + \Delta x)}{2} + lB\left(\frac{1}{l}\right)\left(f(S_j) - f(x_j) - \frac{S_j - x_j}{\Delta x}(f(x_j + \Delta x) - f(x_j))\right), \tag{27}$$

where $S_j \sim \text{Uniform}(x, x + \Delta x)$.

Python Code 2.4.4 (implementation of (2.4.3))
We implement (2.4.3) for $\int_0^1 e^s ds$.

```
from random import random as U
from math import exp
import numpy as np
def f(x): return exp(x)
def trapezium(n): return sum((f(j/n)+f((j+1)/n))/2
    for j in range(n))/n
def MCtrapezium(n, l=100):
    sol = 0
    for j in range(n):
        if U()*l < 1:
            x, xx = j/n, (j+1)/n
            S = x + U()*(xx-x)  # \sim Uniform(x,xx)
            sol += l*(f(S)-f(x)-(S-x)*(f(xx)-f(x))*n)/n
    return sol+trapezium(n)
def exact(a, b): return exp(b)-exp(a)
def error(s): return (s-exact(0, 1))/exact(0, 1)
print(f"  error:{error(trapezium(10000))}")
print(f"MCerror:{error(MCtrapezium(10000,100))}")
#   error:8.333344745642098e-10
# MCerror:-1.5216231703870405e-10
```

Figure 3 suggest that the order of convergence of RMSE of the MC trapezoidal rule is better by 0.5 as the normal trapezoidal rule. The MC trapezoidal rule has on average $\frac{1}{l}$ more function calls than the normal trapezoidal rule. For the composite rule with n intervals, there are Binomial$(n, \frac{1}{l})$ additional function calls (repeated Bernoulli experiments). So they use up to a constant the same amount of function calls. This implies an advantage in IBC.

Theorem 2.4.5 (RMSE Composite Trapezoidal MC Rule)
The composite trapezoidal MC rule wth n intervals for a twice differentiable function has

$$\text{RMSE} = O\left(\frac{1}{n^{2.5}}\right). \tag{28}$$

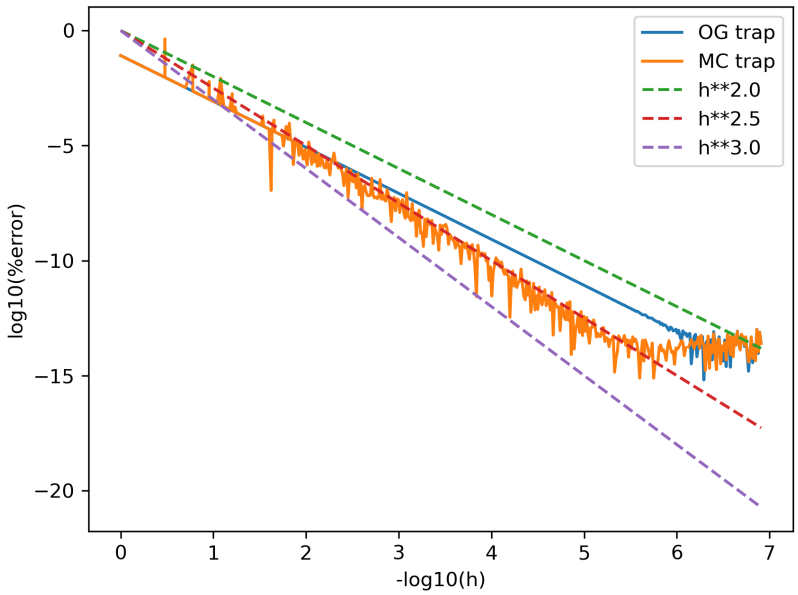

Figure 3: Log-log plot of the error of (2.4.3) for $\int_0^1 e^s ds$ with $l = 100$. At floating point accuracy, the convergence ceases.

The proof uses lemma 2.4.2 and is similar to the proof of the normal trapezoidal rule with the main difference being the accumulation of "local truncation" errors into "global truncation" error. Normally there is a loss of one order but the MC trapezoidal loses only a half order because the accumulation happens in variance instead of bias.

$$\sqrt{\mathrm{Var}\left(\sum_{j=1}^{n} h^3 U_j^2\right)} = h^3 \sqrt{\sum_{j=1}^{n} \mathrm{Var}(U_j^2)} \tag{29}$$

$$= h^3 \sqrt{n \mathrm{Var}(U^2)} \tag{30}$$

$$= O(h^{2.5}). \tag{31}$$

Note that the meaning of a bound on the error that behaves as $O(h^2)$ and a bound on the RMSE that behaves as $O(h^2)$ is different. A bound on the error implies a bound on the RMSE, but the reverse is not true.

Related Work 2.4.6 (Monte Carlo Trapezoidal Rule)
With Stein's paradox it is always possible to bias the composite MC trapezoidal rule to achieve lower RMSE. The optimal IBC for RMSE for some smoothness classes are known see [HN01].

2.5 Unbiased Non-Linearity

In this subsection, we present techniques for handling polynomial non-linearity. The backbone for this is using independent samples $y^2 \cong Y_1 Y_2$ with Y_1 independent of Y_2 and $Y_1 \cong Y_2 \cong y$.

Example 2.5.1 ($y' = y^2$)
Consider the following ODE:

$$y' = y^2, \quad y(1) = -1. \tag{32}$$

The solution to this equation is given by $y(t) = -\frac{1}{t}$. By integrating both sides of equation (32), we obtain the following integral equation:

$$y(t) = -1 + \int_1^t y(s)y(s)ds. \tag{33}$$

To estimate the recursive integral in equation (32), we use i.i.d. $Y_1, Y_2 \sim Y$ in following RRVE:

$$y(t) \cong Y(t) = -1 + (t-1)Y_1(S)Y_2(S), \tag{34}$$

where $S \sim \text{Uniform}(1, t)$. Branching RRVEs are typical when dealing with non-linearity.

Python Code 2.5.2 (implementation (2.5.1))

```
from random import random as U
def Y(t):
    if t>2: raise Exception("doesn't support t>2")
    S = U()*(t-1)+1
    # Y(u)**2 != Y(u)*Y(u) !!!
    return -1 + Y(S)*Y(S) if U()<t-1 else -1
def y(t, nsim): return sum(Y(t) for _ in range(nsim))/nsim
print(f"y(2) approx {y(2,10**3)}")
# y(2) approx -0.488
```

In this implementation $Y(t)$ only takes values $\{-1, 0\}$.

Example 2.5.3 ($e^{E[X]}$)
$e^{\int x(s)ds}$ is a common expression encountered when studying ODEs. In this example, we demonstrate how you can generate unbiased estimates of $e^{E[X]}$ with simulations of X. The Taylor series of e^x is:

$$e^{E[X]} = \sum_{n=0}^{\infty} \frac{E^n[X]}{n!} \tag{35}$$

$$= 1 + \frac{1}{1}E[X]\left(1 + \frac{1}{2}E[X]\left(1 + \frac{1}{3}E[X]\,(1 + ...)\right)\right). \tag{36}$$

Change the fractions of equation (36) to Bernoulli processes and replace all X with independent X_j with $E[X] = E[X_i]$.

$$e^{E[X]} = E\left[1 + B\left(\frac{1}{1}\right)E[X_1]\left(1 + B\left(\frac{1}{2}\right)E[X_2]\left(1 + B\left(\frac{1}{3}\right)E[X_3]\,(1 + ...)\right)\right)\right] \tag{37}$$

$$= E\left[1 + B\left(\frac{1}{1}\right)X_1\left(1 + B\left(\frac{1}{2}\right)X_2\left(1 + B\left(\frac{1}{3}\right)X_3\,(1 + ...)\right)\right)\right] \tag{38}$$

What is inside the expectation is something that we can simulate with simulations of X_j.

Related Work 2.5.4 (example 2.5.3)

NVIDIA has a great paper on optimizing example 2.5.3 [Ket+21].

2.6 Recursion

In this subsection, we discuss recursion-related techniques.

Technique 2.6.1 (coupled recursion)

The idea behind coupled recursion is sharing recursion calls of multiple RRVEs for simulation. This does make them dependent.

Example 2.6.2 (coupled recursion)

Consider calculating the sensitivity of following ODE to a parameter a:

$$y' = ay, y(0) = 1 \Rightarrow \tag{39}$$

$$\partial_a y' = y + a\partial_a y \tag{40}$$

Turn (39) and (40) into RRVEs. To emphasize that they are coupled, that they should recurse together we write them in a matrix equation:

$$\begin{bmatrix} Y(t) \\ \partial_a Y(t) \end{bmatrix} = X(t) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} a & 0 \\ 1 & a \end{bmatrix} X(Ut). \tag{41}$$

Observe how this eliminates the additive branching recursion present in equation (40).

Python Code 2.6.3 (implementation of (41))

```python
from random import random as U
import numpy as np
def X(t, a):  # only supports t<1
    q, A = np.array([1, 0]), np.array([[a, 0], [1, a]])
    return q + A @ X(U()*t, a) if U() < t else q
def sol(t, a, nsim): return sum(X(t, a) for _ in
   range(nsim))/nsim
print(f"x(1,1) = {sol(1,1,10**3)}")
# x(1,1) = [2.7179 2.7104]
```

Related Work 2.6.4 (coupled recursion)

Example 2.6.2 is inspired by [VSJ21]. [VSJ21] propose an efficient unbiased back-propagation algorithm for rendering.

Technique 2.6.5 (recursion in recursion)

Recursion in recursion is like proving an induction step of an induction proof with induction. Recursion in recursion uses an inner recursion in the outer recursion.

Related Work 2.6.6 (recursion in recursion)

Beautiful examples of recursion in recursion are the next flight variant of WoS in [Saw+22] and epoch-based algorithms in optimization [GH21].

Most programming languages do support recursion, but it often comes with certain limitations such as maximum recursion depth and potential performance issues. There are multiple ways to implement recursion we will discuss tail recursion and do an example using a stack.

Technique 2.6.7 (non-branching tail recursion)

Tail recursion involves reordering all operations so that almost no operation needs to happen after the recursion call. This allows us to return the answer without retracing all steps when we reach the last recursion call and it can achieve similar speeds to a forward implementation.

The non-branching recursion presented in the RRVEs can be implemented with tail recursion due to the associativity of all operations ($(xy)z = x(yz)$) involved.

Python Code 2.6.8 (tail recursion on (41))

We implement (41) but this time with tail recursion. We collect addition operations in a vector *sol* and multiplication in a matrix W. W may be referred to as accumulated weight or throughput.

```
1  from random import random as U
2  import numpy as np
3  def X(t, a) -> np.array:
4      q, A = np.array([1.0, 0.0]), np.array([[a, 0.0], [1.0, a]])
5      sol, W = np.array([1.0, 0.0]), np.identity(2)
6      while U() < t:
7          W = W @ A if t < 1 else t * W @ A
8          sol += W @ q
9          t *= U()
10     return sol
11 def sol(t, a, nsim): return sum(X(t, a) for _ in
       range(nsim))/nsim
12 print(f"x(1,1) = {sol(1,1,10**3)}")
13 # x(1,1) = [2.7198 2.7163]
```

Tail recursion is not always desirable as it discards intermediate values of the recursion calls and can increase computational cost. To retain some of these intermediate values, it is possible to use tail recursion partially. In example 2.6.8 it would be more efficient to avoid matrix multiplication on line 7. An alternative to tail recursion is implementing the recursion with a stack.

Python Code 2.6.9 (stack recursion on (41))

We implement (41) but this time with a stack. We want to avoid matrix multiplication and only use matrix-vector multiplications. To do this on (41) we need to know $X(Ut)$ when it doesn't get Russian rouletted away. If we sample Ut we can recurse on our reasoning until Russian roulette termination so we need the path of all the samples.

```
1  from random import random as U
2  from collections import deque
3  import numpy as np
4  def sample_path(t):
5      res = deque([t])
6      while U() < t:
7          t *= U()
```

```
        res.append(t)
    return res
def X(t, a) -> np.array:
    q, A = np.array([1.0, 0.0]), np.array([[a, 0.0], [1.0, a]])
    X, path = np.zeros(2), sample_path(t)
    while path:
        t = path[-1]
        X = q + (A @ X if t < 1 else t * A @ X)
        path.pop()
    return X
def sol(t, a, nsim): return sum(X(t, a) for _ in
   range(nsim))/nsim
print(f"x(1,1) = {sol(1,1,10**3)}")
# x(1,1) = [2.721  2.725]
```

3 Ordinary Differential Equations

3.1 Green Functions

In this subsection, we discuss informally how to turn ODEs into integral equations mainly by example. Our main tool for this are green functions. Prior to discussing green functions, we offer several examples.

Example 3.1.1 ($y' = y$ average condition)
We will solve the equation:

$$y' = y, \tag{42}$$

but this time with a different condition:

$$\int_0^1 y(s)ds = e - 1. \tag{43}$$

The solution to this equation remains the same: $y(t) = e^t$. We define the corresponding source green function $G(t, x)$ for y' and this type of condition as follows:

$$G' = \delta(x - t), \quad \int_0^1 G(s, x)ds = 0. \tag{44}$$

Solving this equation yields:

$$G(t, x) = H(t - x) + x - 1. \tag{45}$$

It is worth noting that we could have chosen a different green function corresponding to a different linear differential operator.
At this point, it may not be clear, but using this green function, we can form the following integral equation for (42):

$$y(t) = e - 1 + \int_0^1 G(t, s)y(s)ds. \tag{46}$$

Converting equation (46) into a RRVE using RMC, we obtain:

$$Y(t) = e - 1 + 2B\left(\frac{1}{2}\right) Y(S)(H(t-S) + S - 1), \tag{47}$$

where $S \sim U$. We will skip the Python implementation of equation (47) as it does not provide any new information. Instead, we will plot realizations of equation (47) in Figure 4.

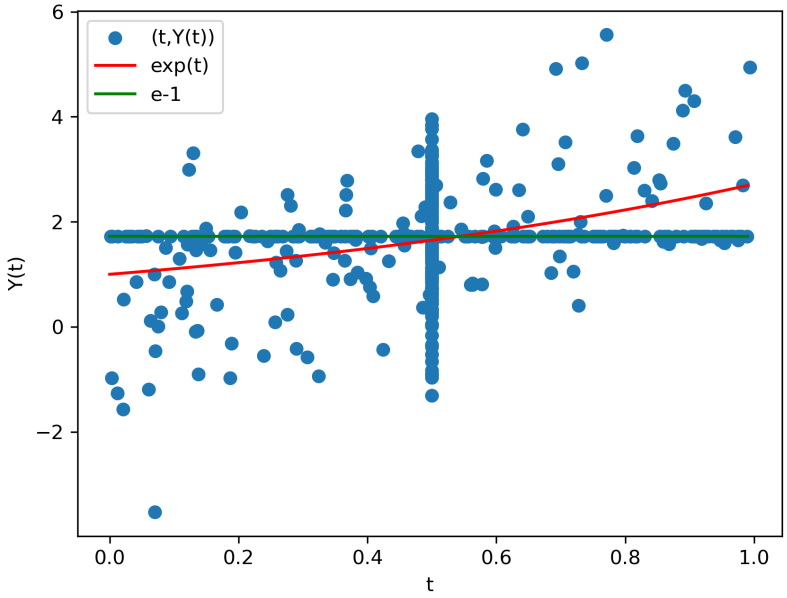

Figure 4: Recursive calls of (47) when calling $Y(0.5)$ 300 times. Points accumulate on the green line due to the Russian roulette, and at $t = 0.5$ because it is the starting value of the simulation.

Definition 3.1.2 (green function)
Vaguely speaking, we define the green function as a type of kernel function used to solve linear problems with linear conditions. The green function is the kernel that we place in front of the linear conditions or the source term, which we integrate over to obtain the solution. The green function possesses the property of satisfying either null linear conditions and a Dirac delta source term, or vice versa.

Related Work 3.1.3 (green function)
Our notion of green function is similar to that in [HGM01].

3.2 Fredholm Integral Equations

TODO:

- explain better the graph for coupled splitting
- add better explaination to coupled splitting

The integral equations obtained in the previous subsection are Fredholm integral equations of the second kind. In this subsection, we introduce a technique called coupled splitting.

Definition 3.2.1 (Fredholm equation of the second kind)
A Fredholm equation of the second kind for φ is of the following form:

$$\varphi(t) = f(t) + \lambda \int_a^b K(t,s)\varphi(s)ds. \tag{48}$$

Given the kernel $K(t,s)$ and $f(t)$.

If both K and f satisfy certain regularity conditions, then for sufficiently small λ, it is relatively straightforward to establish the existence and uniqueness of solutions using a fixed-point argument.

Example 3.2.2 (Dirichlet $y'' = y$)
The following ODE will be the main testing example for boundary value problems:

$$y'' = y, \quad y(b_0), y(b_1). \tag{49}$$

The green functions corresponding to y'' and Dirichlet conditions are:

$$P(t,x) = \begin{cases} \frac{b_1 - t}{b_1 - b_0} & \text{if } x = b_0 \\ \frac{t - b_0}{b_1 - b_0} & \text{if } x = b_1 \end{cases}, \tag{50}$$

$$G(t,s) = \begin{cases} -\frac{(b_1 - t)(s - b_0)}{b_1 - b_0} & \text{if } s < t \\ -\frac{(b_1 - s)(t - b_0)}{b_1 - b_0} & \text{if } t < s \end{cases}. \tag{51}$$

Straight from these green functions you get the following integral equation and RRVE:

$$y(t) = P(t,b_0)y(b_0) + P(t,b_1)y(b_1) + \int_{b_0}^{b_1} G(t,s)y(s)ds, \tag{52}$$

$$Y(t) = P(t,b_0)y(b_0) + P(t,b_1)y(b_1) + lB\left(\frac{1}{l}\right)(b_1 - b_0)G(t,S)y(S), \tag{53}$$

where $l \in \mathbb{R}$ the Russian roulette rate is and $S \sim \text{Uniform}(b_1, b_0)$.

Example 3.2.3 (coupled splitting on (3.2.2))
In addition to normal splitting (see Definition 2.3.6), we can also split the domain in Equation (52) as follows:

$$y(t) = P(t,b_0)y(b_0) + P(t,b_1)y(b_1) + \frac{1}{2}\int_{b_0}^{b_1} G(t,s)y(s)ds + \frac{1}{2}\int_{b_0}^{b_1} G(t,s)y(s)ds, \tag{54}$$

$$y(t) = P(t,b_0)y(b_0) + P(t,b_1)y(b_1) + \int_{b_0}^{\frac{b_1+b_0}{2}} G(t,s)y(s)ds + \int_{\frac{b_1+b_0}{2}}^{b_1} G(t,s)y(s)ds. \tag{55}$$

By coupling, we can eliminate the additive branching recursion in the RRVEs corresponding to Equations (54) and (55). This results in the following RRVE:

$$X(t_1,t_2) = \begin{bmatrix} P(t_1,b_0) & P(t_1,b_1) \\ P(t_2,b_0) & P(t_2,b_1) \end{bmatrix} \begin{bmatrix} y(b_0) \\ y(b_1) \end{bmatrix} + W \begin{bmatrix} G(t_1,S_1) & G(t_1,S_2) \\ G(t_2,S_1) & G(t_2,S_2) \end{bmatrix} X(S_1,S_2), \tag{56}$$

where W the right weighting matrix is (see code (3.2.4)), and S_1 and S_2 can be chosen in various ways.

Python Code 3.2.4 (implementation of (56))
We implemented equation (56) in example (3.2.3) with recursion but in this case, it is also possible to implement it forwardly.

```
from random import random as U
from math import exp
import numpy as np
def Pb0(t, b0, b1): return (b1-t)/(b1-b0)
def Pb1(t, b0, b1): return (t-b0)/(b1-b0)
def G(t, s, b0, b1): return - (b1-s)*(t-b0)/(b1-b0) if t < s
   else - (b1-t)*(s-b0)/(b1-b0)
def X(T, y0, y1, b0, b1):
    yy = np.array([y0, y1])
    bb = np.diag([(b1-b0)/len(T)]*len(T))
    PP = np.array([[Pb0(t, b0, b1), Pb1(t, b0, b1)] for t in T])
    sol = PP @ yy
    l = 1.2  # russian roulette rate
    if U()*l < 1:
        u = U()
        SS = [b0+(u+j)*(b1-b0)/len(T) for j in range(len(T))]
        GG = np.array([[G(t,S,b0,b1) for S in SS] for t in T])
        sol += l*GG @ bb @ X(SS, y0, y1, b0, b1)
    return sol
```

Example (3.2.3) does not exploit the locality and smoothness of the problem. We conjecture that coupled splitting can be particularly valuable when dealing with linear Fredholm equations of the second kind, especially in scenarios where MC integration surpasses traditional integration methods. This holds true for high-dimensional problems, non-smooth kernels, and challenging domains. We conjecture that employing coupled splitting in such cases can yield favorable results.

Related Work 3.2.5 (coupled splitting)
Coupled splitting is partly inspired by how [SM09] reduces variance by using bigger submatrices. Reusing samples for walk on sphere gets discussed in [Mil+23] and [BP23].

Figure 5 resembles fixed-point iterations, leading us to hypothesize that coupled splitting can achieve convergence in most cases where a fixed-point argument holds true and the convergence speed is very similar to fix-points methods until the accuracy of the stochastic approximation of the operator is reached (the approximate operator bottleneck). The approximation of the operator can be improved by increasing coupled splitting amount when approaching the bottleneck. Alternatively when reaching the bottleneck it is possible to rely on MC convergence.

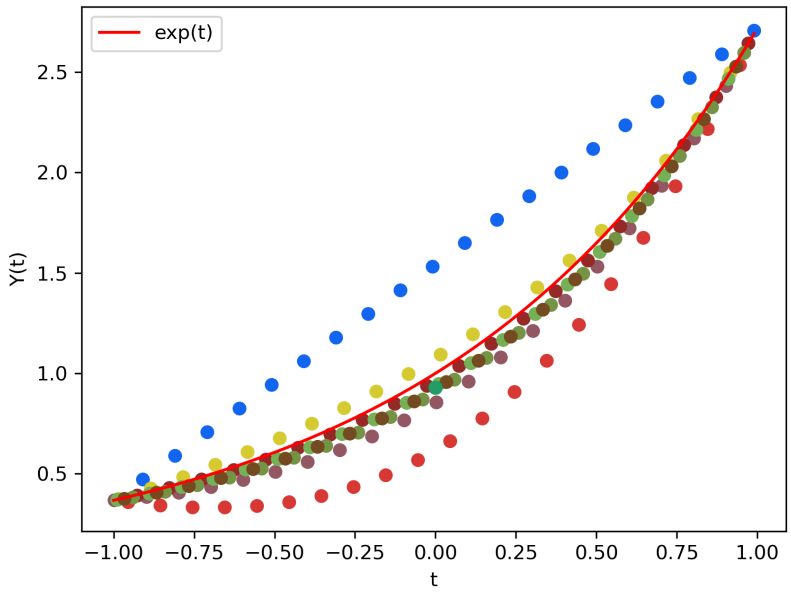

Figure 5: Recursive calls of equation (56) when calling $X(0)$ once, with a split size of 20, S_j are coupled such they are equally spaced and the coupling is colored. The initial conditions for this call are $y(-1) = e^{-1}$ and $y(1) = e^1$, with Russian roulette rate $l = 1.2$.

Related Work 3.2.6 (convergence coupled splitting)
See [GH21] for a discussion on the convergence of recursive stochastic algorithms.

Coupled splitting was originally motivated to help with convergence. It didn't increase the convergence domain for the example shown in Figure 6.

3.3 Initial Value Problems

Classic IVP solvers rely on shrinking the time steps for convergence. In this subsection we build up to Recursion in Recursion MC (RRMC) for IVPs that tries to emulate this behavior.

Example 3.3.1 (RRMC $y' = y$)
We demonstrate RRMC for IVPs with

$$y' = y, \quad y(0) = 1. \tag{57}$$

Imagine we have a time-stepping scheme $(t_n), \forall n : t_{n-1} < t_n$ then the following integral equations hold:

$$y(t) = y(t_n) + \int_{t_n}^{t} y(s)ds, \quad t > t_n. \tag{58}$$

Turn these in the following class of RRVEs:

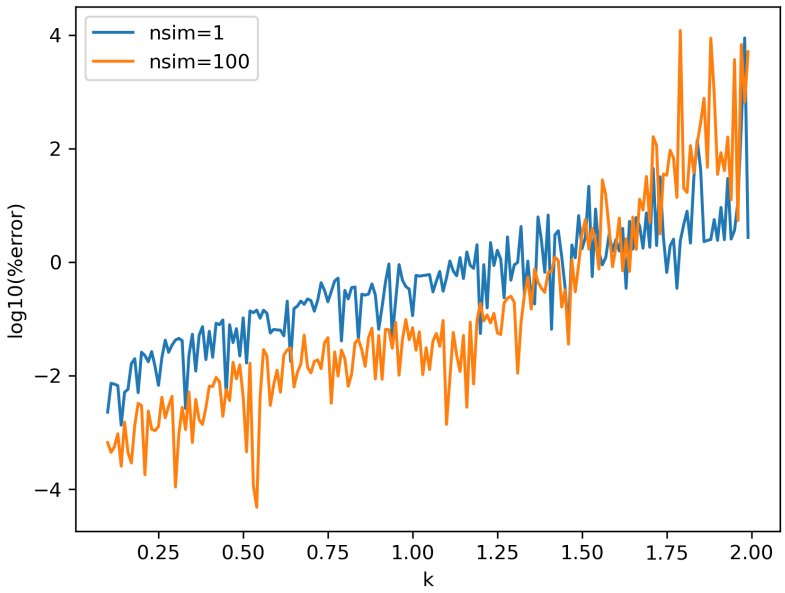

Figure 6: The logarithmic percentage error of $Y(0)$ for (53), with $l = 1.2$ and initial conditions $y(-k) = e^{-k}$ and $y(k) = e^{k}$, displays an exponential increase until approximately $k = 1.5$, beyond which additional simulations fail to reduce the error, indicating that the variance doesn't exist.

$$Y_n(t) = y(t_n) + (t - t_n)Y_n((t - t_n)U + t_n), \quad t > t_n. \tag{59}$$

A problem with these RRVEs is that we do not know $y(t_n)$. Instead, we can replace it with an unbiased estimate y_n which we keep frozen in the inner recursion:

$$Y_n(t) = y_n + (t - t_n)Y_n((t - t_n)U + t_n), \quad t > t_n \tag{60}$$

$$y_n = \begin{cases} Y_{n-1}(t_n) & \text{if } n \neq 0 \\ y(t_0) & \text{if } n = 0 \end{cases}. \tag{61}$$

We refer to equation (60) as the inner recursion and equation (61) as the outer recursion of the recursion in recursion.

Python Code 3.3.2 (implementation of (3.3.1))

```
from random import random as U
def Y_in(t, tn, yn, h):
    S = tn + U()*(t-tn)  # \sim Uniform(T,t)
    return yn + h*Y_in(S, tn, yn, h) if U() < (t-tn)/h else yn
def Y_out(tn, h): # h is out step size
    TT = tn-h if tn-h > 0 else 0
    return Y_in(tn, TT, Y_out(TT, h), h) if tn > 0 else 1
```

We measured the convergence speed of example (3.3.1) to be $O\left(\frac{h^{1.5}}{\sqrt{\text{nsim}}}\right)$. While a 1.5 order of convergence is commendable for an unbiased method, it raises the question

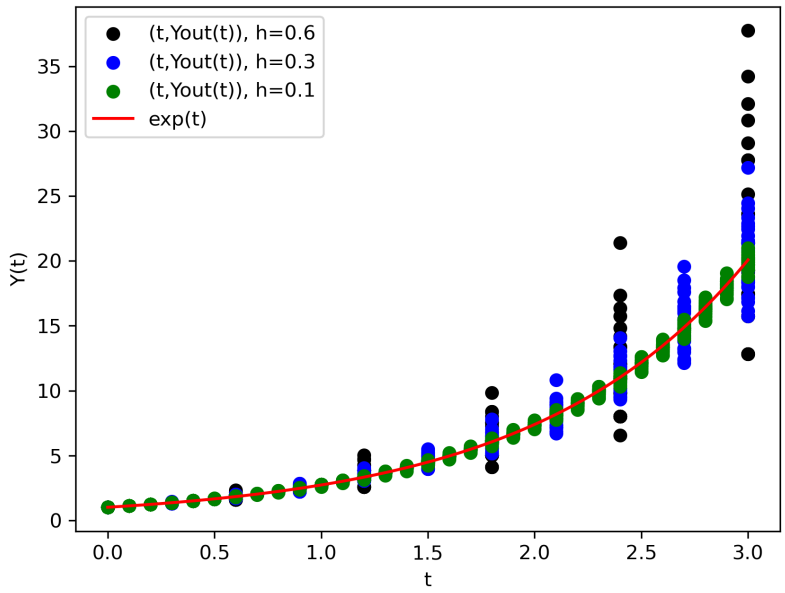

Figure 7: Recursive calls of equation (61) when calling $Y_{out}(3, h)$ 30 times for different h.

of how to attain even higher convergence orders. It is possible to emulate classical methods to achieve a higher order of convergence. This can be accomplished by eliminating lower order terms by using control variates, as demonstrated in the MC trapezoidal rule (see 2.4.3).

Example 3.3.3 (CV RRMC $y' = y$)
Let us control variate example (3.3.1).

$$y(t) = y(t_n) + \int_{t_n}^{t} y(s)ds, \quad t > t_n. \tag{62}$$

We build a control variate with a lower-order approximation of the integrand:

$$\begin{aligned} y(s) &= y(t_n) + (s - t_n)y'(t_n) + O((s - t_n)^2) && (63)\\ &\approx y(t_n) + (s - t_n)f(y(t_n), t_n) && (64)\\ &\approx y(t_n) + (s - t_n)\left(\frac{y(t_n) - y(t_{n-1})}{t_n - t_{n-1}}\right) && (65)\\ &\approx y(t_n)(1 + s - t_n). && (66) \end{aligned}$$

Using the last one as a control variate for the integral:

$$y(t) = y(t_n) + \int_{t_n}^{t} y(s)ds \tag{67}$$

$$= y(t_n) + \int_{t_n}^{t} y(s) - y(t_n)(1 + s - t_n) + y(t_n)(1 + s - t_n)ds \tag{68}$$

$$= y(t_n)\left(1 + (1 - t_n)(t - t_n) + \frac{t^2 - t_n^2}{2}\right) + \int_{t_n}^{t} y(s) - y(t_n)(1 + s - t_n)ds. \tag{69}$$

We will not discuss turning this into an RRVE nor the implementation. The implementation is very similar to (3.3.6) and Figure 8 is a convergence plot for this example.

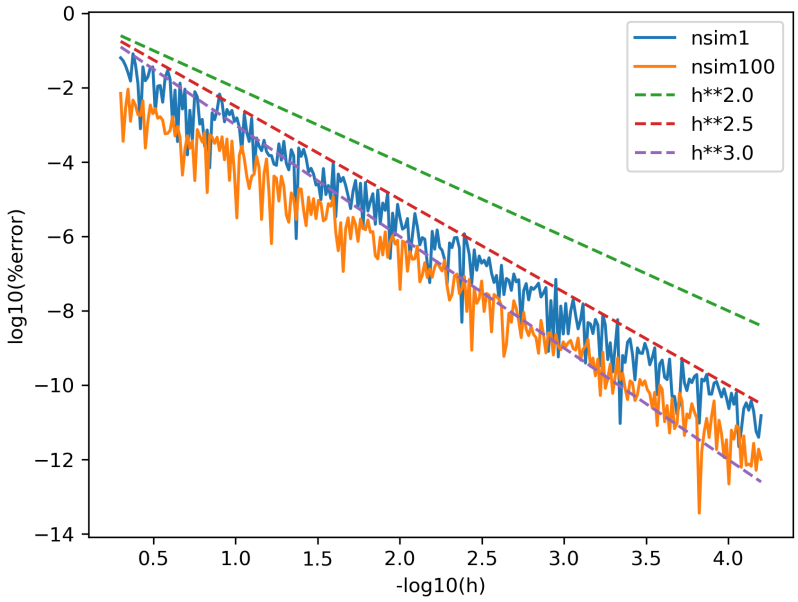

Figure 8: Log-log plot of the error for example (3.3.3) at $Y(10)$.

Related Work 3.3.4 (CV RRMC)

[Dau11] similarly uses control variates to achieve a higher order of convergence.

RRMC is biased for our approach to non-linear problems. The inner recursions are correlated because they use the same information from the outer recursions, this does not mean that reducing root-mean-square error by splitting does not work, you just have to be careful with the bias. We conjecture that the bias in RRMC converges faster than the variance when decreasing h.

Example 3.3.5 (nonlinear RRMC IVP)

Consider the following IVP:

$$y' = y^2 - t^4 + 2t, \quad y(0) = 0. \tag{70}$$

The exact solution to this problem is given by $y(t) = t^2$. We can express the solution as an integral equation:

$$y(t) = y(t_n) + \int_{t_n}^{t} y^2(s)ds - \frac{t^5 - t_n^5}{5} + (t^2 - t_n^2). \tag{71}$$

By control variating $y^2(s)$ up to the second order using Taylor expansion, we have:

$$\begin{aligned} y^2(t) &\approx y^2(t_n) + 2(t - t_n)y(t_n)y'(t_n) + ((t - t_n)y'(t_n))^2 + O((t - t_n)^2) && (72)\\ &\approx y^2(t_n) + 2(t - t_n)y(t_n)y'(t_n) + O((t - t_n)^2) && (73) \end{aligned}$$

Integrating the control variate yields:

$$\begin{aligned} &\int_{t_n}^{t} y^2(t_n) + 2(s - t_n)y(t_n)y'(t_n)ds && (74)\\ &\quad = (t - t_n)y^2(t_n) + 2\left(\frac{t^2 - t_n^2}{2} - t_n(t - t_n)\right)y(t_n)y'(t_n). && (75) \end{aligned}$$

We implement this example in (3.3.6).

Python Code 3.3.6 (implementation of (3.3.5))

```
from random import random as U
def Y_in(t, tn, yn, dyn, h, l):
    sol = yn  # initial conditon
    sol += t**2-tn**2 - (t**5-tn**5)/5  # source
    sol += (t-tn)*yn**2  # 0 order
    sol += 2*((t**2-tn**2)/2 - tn*(t-tn))*yn*dyn  # 1 order
    if U()*l < (t-tn)/h:
        S = tn + U()*(t-tn)  # \sim Uniform(T,t)
        sol += l*h*(Y_in(S, tn, yn, dyn, h, l)*
            Y_in(S, tn, yn, dyn, h, l) - yn**2-2*(S-tn)*yn*dyn)
    return sol
def Y_out(t, h, l):
    yn, tn = 0, 0
    while tn < t:
        tt = tn+h if tn+h < t else t
        dyn = yn**2 - tn**4+2*tn
        yn = Y_in(tt, tn, yn, dyn, tt-tn, l)
        tn = tt
    return yn
```

4 Limitations and Future Work

We have not dedicated significant attention, if any at all, to the convergence analysis of RMC methods for Fredholm equations. Figure 6 illustrates that arbitrary RMC do not converge.

Similarly to classic methods, RRMC struggles with big negative coefficients in front of the recursive parts which may appear in stiff problems. In addition to the challenges with stiff problems, we have not conducted a comparison between RRMC and traditional methods, and we do not anticipate that RRMC in this form outperforms other classic methods. The reason for this is that RRMC involves many function calls in the inner recursion without updating the current control variate. Proper testing requires precise tuning of the MC techniques utilized. We do believe that RRMC could potentially offer advantages in atypical scenarios. In the following example, we examine an ODE with a random parameter.

Example 4.0.1 (random ODEs)
Consider the problem given by the initial value problem

$$Y' = AY, \quad Y(0) = 1, \quad A \sim \text{Uniform}(0, 1). \tag{76}$$

The solution to this problem is given by $Y(t) = e^{tA}$, which is a RV. In order to compute expectations of functions of $Y(t)$, we use unbiased estimates X of the simulations of the solution.
Given an analytic function f, we can obtain $E[f(Y(t))]$ by conditioning on the value of A and using the law of total expectation:

$$E_A[f(Y(t))] = E_A[f(Y(t)) \mid A] \tag{77}$$
$$= E_A[f(E_X[X(t, A)]) \mid A]. \tag{78}$$

To estimate $f(E_X[X(t, A)])$, we use the approach outlined in (2.5.3). For the specific example of (76), we can compute the first two moments of $Y(t)$ as

$$E_A[Y(t)] = \frac{e^t}{t} - \frac{1}{t}, \tag{79}$$
$$E_A[Y^2(t)] = \frac{e^{2t}}{2t} - \frac{1}{2t}. \tag{80}$$

Python Code 4.0.2 (implementation of (4.0.1))

```
from random import random as U
from math import exp
def X(t, a):
    if t < 1: return 1+a*X(U()*t, a) if U() < t else 1
    return 1+t*a*X(U()*t, a)
def eY(t): return X(t, U())
def eY2(t):
    A = U()
    return X(t, A)*X(t, A)
t, nsim = 3, 10**4
sol = sum(eY(t) for _ in range(nsim))/nsim
sol2 = sum(eY2(t) for _ in range(nsim))/nsim
s = exp(t)/t - 1/t                # analytic solution
s2 = exp(2*t)/(2*t) - 1/(2*t)     # analytic solution
print(f"E(Y({t}))   is approx {sol},%error = {(sol - s)/s}")
print(f"E(Y^2({t})) is approx {sol2},%error = {(sol2 - s2)/s2}")
# E(Y(3))   is approx 6.5683, %error = 0.0324
# E(Y^2(3)) is approx 64.5843, %error = -0.0370
```

4.1 Heat Equation

In this subsection, we introduce the relation between the heat equation and Brownian motion.

Definition 4.1.1 (1D heat equation Dirichlet)
We define the 1D heat equation for u on connected domain Ω with Dirichlet boundary conditions the following way:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}. \tag{81}$$

Given $u(x,t) = \psi(x,t), \forall (x,t) \in \partial\Omega : t < \sup\{t|(x,t) \in \Omega\}$.

Lemma 4.1.2 (Brownian motion and the heat equation)
For problem (4.1.1) if $|\psi|$ is bounded then there holds:

$$u(x,t) = E[\psi(Y_\tau, \tau)|Y_t = x]. \tag{82}$$

With $dY_s = dW_{-s}, \tau = \sup\{s|(Y_s, s) \notin \Omega\}$.

Proof. Discretize the heat equation with a regular rectangular mesh that includes (x,t) with equally spaced intervals over space and time $(\Delta x, \Delta t)$ with the corresponding difference equation:

$$\frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}. \tag{83}$$

Isolate $u(x,t)$:

$$u(x,t) = \frac{\Delta t}{2\Delta t + \Delta x^2}\left(u(x+\Delta x,t) + u(x-\Delta x,t)\right) + \frac{\Delta x^2}{2\Delta t + \Delta x^2}\left(u(x,t-\Delta t)\right). \tag{84}$$

Because $u(x+\Delta x,t) \approx u(x-\Delta x,t) \approx u(x,t-\Delta t) \approx$ right-hand side of equation (84) we may Russian roulette to remove branching recursion and generate a recursion path instead of a tree.

$$Z(x,t) = \begin{cases} \psi(\text{argmin}_{b\in\partial\Omega}||(x,t)-b||) & \text{when } (x,t) \notin \Omega \\ \begin{cases} Z(x+\Delta x,t) & \text{with chance } \frac{\Delta t}{2\Delta t+\Delta x^2} \\ Z(x-\Delta x,t) & \text{with chance } \frac{\Delta t}{2\Delta t+\Delta x^2} \\ Z(x,t-\Delta t) & \text{with chance } \frac{\Delta x^2}{2\Delta t+\Delta x^2} \end{cases} & \text{else .} \end{cases} \tag{85}$$

This is a RRVE, Z has finite variance because $|\psi|$ is bounded and $E[Z(x,t)]$ is the solution to the discretized heat equation. Taking the limit makes the discrete solution converge to the real solution. For (85) the limit makes the recursion path go to Brownian motion Y_t.

$$Z(x,t) \to \psi(Y_\tau, \tau). \tag{86}$$

Finishing the proof. □

Related Work 4.1.3
(4.1.2) is a subcase of the Feynman-Kac formula. For a proof and an in-depth discussion of the Feynman-Kac formula see [Øks03].

Abstract

Deze scriptie onderzoekt recursieve Monte Carlo voor het oplossen van lineaire gewone differentiaalvergelijkingen met het oog op partiële differentiaalvergelijkingen. De voorgestelde algoritmes maken gebruik van de geschikte combinatie van Monte Carlo technieken. Deze Monte Carlo technieken worden geïntroduceerd met voorbeelden en code.

References

[BP23] Ghada Bakbouk and Pieter Peers. "Mean Value Caching for Walk on Spheres". en. In: (2023). Artwork Size: 10 pages Publisher: The Eurographics Association, 10 pages. DOI: 10.2312/SR.20231120. URL: https://diglib.eg.org/handle/10.2312/sr20231120 (visited on 08/01/2023).

[Dau11] Thomas Daun. "On the randomized solution of initial value problems". en. In: *Journal of Complexity* 27.3-4 (June 2011), pp. 300–311. ISSN: 0885064X. DOI: 10.1016/j.jco.2010.07.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S0885064X1000066X (visited on 03/06/2023).

[ES21] S. M. Ermakov and M. G. Smilovitskiy. "The Monte Carlo Method for Solving Large Systems of Linear Ordinary Differential Equations". en. In: *Vestnik St. Petersburg University, Mathematics* 54.1 (Jan. 2021), pp. 28–38. ISSN: 1063-4541, 1934-7855. DOI: 10.1134/S1063454121010064. URL: https://link.springer.com/10.1134/S1063454121010064 (visited on 01/20/2023).

[GH21] Abhishek Gupta and William B. Haskell. *Convergence of Recursive Stochastic Algorithms using Wasserstein Divergence*. en. arXiv:2003.11403 [cs, eess, math, stat]. Jan. 2021. URL: http://arxiv.org/abs/2003.11403 (visited on 01/11/2023).

[HGM01] Chi-Ok Hwang, James A. Given, and Michael Mascagni. "The Simulation–Tabulation Method for Classical Diffusion Monte Carlo". en. In: *Journal of Computational Physics* 174.2 (Dec. 2001), pp. 925–946. ISSN: 00219991. DOI: 10.1006/jcph.2001.6947. URL: https://linkinghub.elsevier.com/retrieve/pii/S0021999101969475 (visited on 12/18/2022).

[HN01] S. Heinrich and E. Novak. *Optimal Summation and Integration by Deterministic, Randomized, and Quantum Algorithms*. en. arXiv:quant-ph/0105114. May 2001. URL: http://arxiv.org/abs/quant-ph/0105114 (visited on 03/19/2023).

[Hua+17] Yipeng Huang et al. "Hybrid analog-digital solution of nonlinear partial differential equations". en. In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. Cambridge Massachusetts: ACM, Oct. 2017, pp. 665–678. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3124550. URL: https://dl.acm.org/doi/10.1145/3123939.3124550 (visited on 03/01/2023).

[Ket+21] Markus Kettunen et al. "An unbiased ray-marching transmittance estimator". en. In: *ACM Transactions on Graphics* 40.4 (Aug. 2021). arXiv:2102.10294 [cs], pp. 1–20. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3450626.3459937. URL: http://arxiv.org/abs/2102.10294 (visited on 06/18/2023).

[Mil+23] Bailey Miller et al. *Boundary Value Caching for Walk on Spheres*. en. arXiv:2302.11825 [cs]. Feb. 2023. URL: http://arxiv.org/abs/2302.11825 (visited on 05/02/2023).

[Øks03] Bernt Øksendal. *Stochastic Differential Equations*. en. Universitext. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003. ISBN: 978-3-540-04758-2 978-3-642-14394-6. DOI: 10.1007/978-3-642-14394-6. URL: http://link.springer.com/10.1007/978-3-642-14394-6 (visited on 05/08/2023).

[Saw+22] Rohan Sawhney et al. "Grid-free Monte Carlo for PDEs with spatially varying coefficients". en. In: *ACM Transactions on Graphics* 41.4 (July 2022), pp. 1–17. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3528223.3530134. URL: https://dl.acm.org/doi/10.1145/3528223.3530134 (visited on 09/17/2022).

[SM09] K. Sabelfeld and N. Mozartova. "Sparsified Randomization Algorithms for large systems of linear equations and a new version of the Random Walk on Boundary method". en. In: *Monte Carlo Methods and Applications* 15.3 (Jan. 2009). ISSN: 0929-9629, 1569-3961. DOI: 10.1515/MCMA.2009.015. URL: https://www.degruyter.com/document/doi/10.1515/MCMA.2009.015/html (visited on 09/17/2022).

[Vea97] Eric Veach. "Robust Monte Carlo Methods for Light Transport Simulation. Ph.D. Dissertation. Stanford University." en. In: *Robust Monte Carlo Methods for Light Transport Simulation.* (1997).

[VSJ21] Delio Vicini, Sébastien Speierer, and Wenzel Jakob. "Path replay backpropagation: differentiating light paths using constant memory and linear time". en. In: *ACM Transactions on Graphics* 40.4 (Aug. 2021), pp. 1–14. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3450626.3459804. URL: https://dl.acm.org/doi/10.1145/3450626.3459804 (visited on 02/17/2023).