Masterproef scriptie

Recursive Monte Carlo for linear ODEs

Auteur: *Isidoor Pinillo Esquivel*

Promotor: *Wim Vanroose*

Academiejaar 2022-2023

Contents

Abstract

This thesis explores unbiased Monte Carlo methods 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.

This version is where we informally self-criticize, mention shortcomings, doubts and intuition. Comments will be in dark green like this one.

1 Introduction

1.1 Introductory Example

The more formal introduction is with Von Neumann series. We maybe should have mentioned Von Neumann series exist.

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

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

By integrating both sides of equation (1), we obtain:

$$y(t) = 1 + \int_0^t y(s)ds. \tag{2}$$

Equation (2) represents a recursive integral equation, specifically a linear Volterra integral equation of the second type. By estimating the recursive integral in equation (2) using Monte Carlo, we derive the following estimator:

$$Y(t) = 1 + ty(Ut), \tag{3}$$

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:

$$Y(t) = 1 + tY(Ut). \tag{4}$$

Equation (4) is a recursive random variable equation. If one were to try to simulate Y with equation (4), it would recurse indefinitely. To overcome this, approximate $Y(t) \approx 1$ near $t = 0$ introducing minimal bias. Later, we will discuss Russian roulette (2.1.1), which can be used as an unbiased stopping mechanism.

Python Code 1.1.1 (implementation of (4))

```python
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.

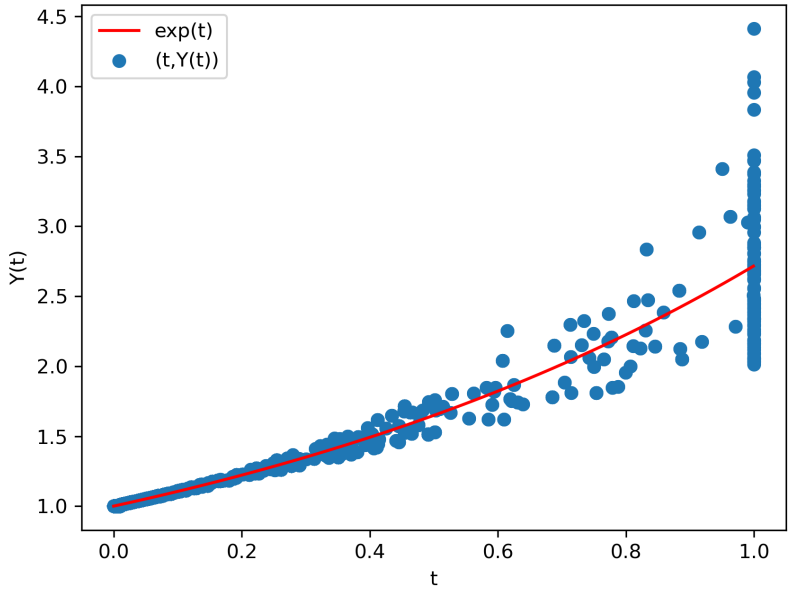

Figure 1: Recursive calls of (1.1.1)

One issue with (1.1.1) is that the variance increases rapidly as t increases. This issue gets resolved later (see (3.3.1)). It is important to note that (1.1.1) retains desirable properties of unbiased Monte Carlo methods, such as being embarrassingly parallel and having simple error estimates.

1.2 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
- FP = First Passage
- $H(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{else} \end{cases}$

- i.i.d. = independent identically distributed
- $U \sim \text{Uniform}(0, 1)$
- 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.

1.3 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 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 focus on the underlying mechanics of these Monte Carlo techniques.

Related Work 1.3.1 (MC for IVPs ODEs)
Monte Carlo methods for initial value problems for systems of ordinary differential equations are little studied. The most significant works are:

- Jentzen and Neuenkirch's 2009 [JN09] paper describes a random Euler scheme for weak smoothness conditions.
- Daun's 2011 [Dau11] paper describes a randomized algorithm that achieves optimal order of convergence by using polynomial extrapolation and control variates.
- Ermakov and Tovstik's 2019 [ET19] and Ermakov and Smilovitskiy's 2021 [ES21] papers study unbiased methods for linear problems and slightly biased methods for nonlinear based on Volterra integral equations.

1.4 Contributions

A significant part of this thesis is dedicated to informally introducing recursive Monte Carlo with plenty of examples. The key contribution is an unbiased Monte Carlo method (3.3.3) for linear initial value problems for systems of ordinary differential equations by using recursion in recursion. This method achieves a higher order of convergence in the time step size that is comparable to explicit deterministic methods.
In the background section, we provide an introduction to Monte Carlo techniques and a short discussion about a Monte Carlo trapezoidal rule. We also discuss coupled recursion and recursion in recursion.

In the ordinary differential equation section, we introduce green functions as a tool to transform ordinary differential equations into integral equations. We then proceed to present our main algorithm for solving linear initial value problems for systems of ordinary differential equations.

In the Brownian motion section, we establish the connection between Brownian motion and the heat equation using recursive Monte Carlo. Additionally, we introduce recursive first passage sampling.

2 Background

2.1 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.

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 2.1.1 (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:

$$X \to \begin{cases} \frac{1}{p}(X - (1-p)Y_1) & \text{if } U < p \\ Y_2 & \text{else} \end{cases}. \tag{5}$$

Example 2.1.2 (Russian roulette)
Let us consider the estimation of $E[Z]$, where Z is defined as follows:

$$Z = U + \frac{f(U)}{1000}. \tag{6}$$

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:

$$\tilde{Z} = U + B\left(\frac{1}{100}\right)\frac{f(U)}{10}. \tag{7}$$

With this modification, $\tilde{Z}$ requires calling f on average once every 100 simulations, significantly reducing the computational burden compared to simulating Z. Although the variance of $\tilde{Z}$ increases slightly compared to Z, it is more efficient.

Related Work 2.1.3 (Russian roulette)
For example (2.1.2) it is 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.
In [Rat+22] they approximate optimal Russian roulette and splitting (RRS) factors with fixed-point iterations to maximize the efficiency of a renderer.

Example 2.1.4 (Russian roulette on (4))
To address the issue of indefinite recursion in equation (4), 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:

$$Y(t) = \begin{cases} 1 + B(t)Y(Ut) & \text{if } t < 1 \\ 1 + tY(Ut) & \text{else} \end{cases}. \tag{8}$$

Python Code 2.1.5 (implementation of (2.1.4))

```
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.

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

$$\bar{X} = \frac{1}{N}\sum_{j=1}^{N} X_j. \tag{9}$$

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

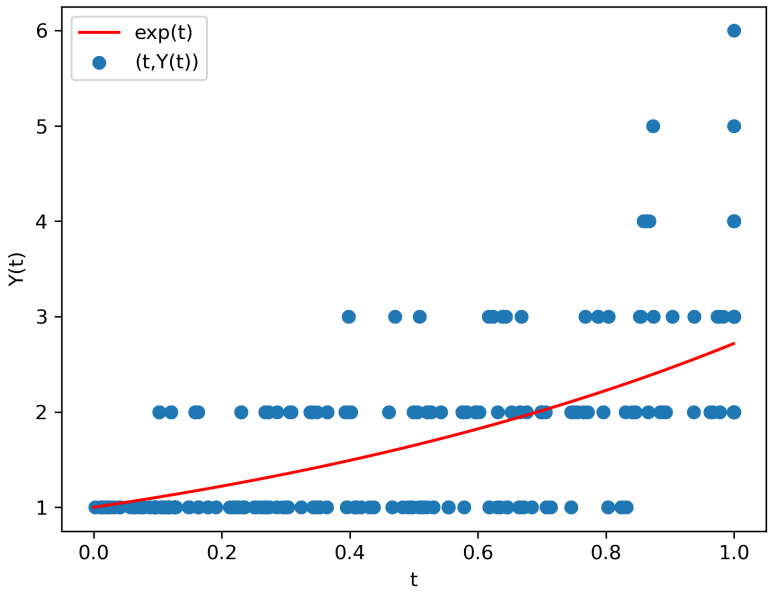

Figure 2: Recursive calls $(t, Y(t))$ of (2.1.5)

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 2.1.7 (splitting on (4))
We can "split" the recursive term of Equation (4) into two parts as follows:

$$Y(t) = 1 + \frac{t}{2}(Y_1(Ut) + Y_2(Ut)), \tag{10}$$

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

Python Code 2.1.8 (implementation of (2.1.7))

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

$$f(X) \to (f - \tilde{f})(X) + E[\tilde{f}(X)]. \tag{11}$$

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 2.1.10 (control variate on (4))
To create a control variate for Equation (4) that effectively reduces variance, we employ the approximation $y(t) \approx 1 + t$ and define the modified recursive term as follows:

$$Y(t) = 1 + t + \frac{t^2}{2} + t(Y(Ut) - 1 - Ut). \tag{12}$$

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 2.1.11 (implementation of (2.1.10))

```
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 2.1.12 (MC modification)
Our favorite work that discusses these techniques is [Vea97]. More interesting MC techniques can be found in rendering.

2.2 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 2.2.1 (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:

$$\int_x^{x+\Delta x} f(s)ds \tag{13}$$

$$= \int_x^{x+\Delta x} \tilde{f}(s)ds + \int_x^{x+\Delta x} f(s) - \tilde{f}(s)ds \tag{14}$$

$$= \Delta x \frac{f(x) + f(x+\Delta x)}{2} + E\left[lB\left(\frac{1}{l}\right)(f(S_x) - \tilde{f}(S_x))\right] \tag{15}$$

$$\approx \Delta x \frac{f(x) + f(x+\Delta x)}{2} \tag{16}$$

$$+ \Delta x l B\left(\frac{1}{l}\right)\left(f(S_x) - f(x) - \frac{S_x - x}{\Delta x}(f(x+\Delta x) - f(x))\right), \tag{17}$$

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

This 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.

Definition 2.2.2 (composite MC trapezoidal rule)

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

$$\int_a^b f(s)ds \approx \Delta x \sum_x \frac{f(x) + f(x + \Delta x)}{2} \tag{18}$$

$$+ lB\left(\frac{1}{l}\right)\left(f(S_x) - f(x) - \frac{S_x - x}{\Delta x}(f(x + \Delta x) - f(x))\right), \tag{19}$$

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

Python Code 2.2.3 (implementation of (2.2.2))

We implement (2.2.2) 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
```

We postulate that this MC composite rule enhances the convergence rate by 0.5 orders compared to the standard composite rule for each dimension, provided that the appropriate conditions of smoothness are met. To substantiate this conjecture, we shall outline our rationale concerning the accumulation of unbiased polynomial errors.

For the sake of simplicity, we reduce the study of the local truncation error to the following expression:

$$\int_0^h s^2 ds = O(h^3). \tag{20}$$

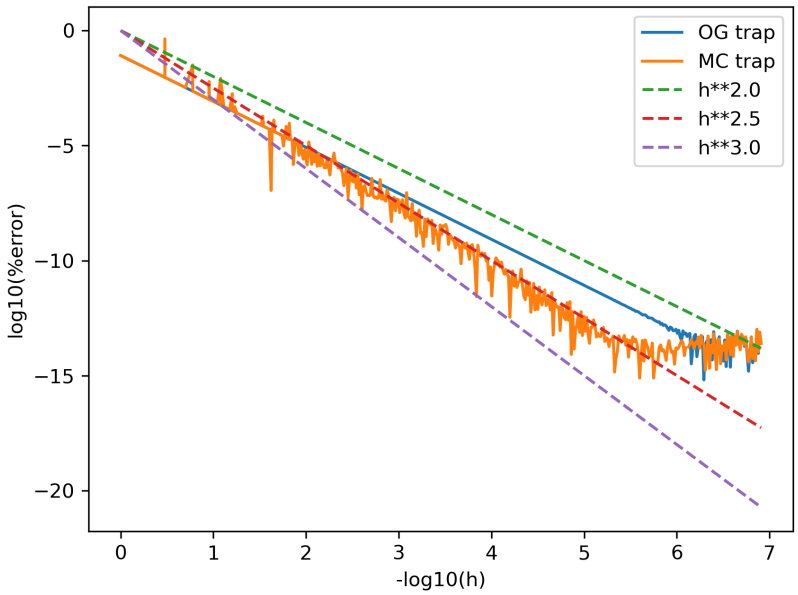

Figure 3: Log-log plot of the error of (2.2.2) for $\int_0^1 e^s ds$ with $l = 100$ at which the additional function calls are negligible. At floating point accuracy, the convergence ceases.

In the standard composite rule, we would drop this term, but in the MC version, we eliminate the bias. As a result, the local truncation error behaves similarly to:

$$\int_0^h s^2 ds - h(hU)^2 = \int_0^h s^2 ds - h^3 U^2 = O(h^3). \tag{21}$$

The main distinction between the standard and MC rule lies in how they accumulate local truncation errors into global truncation error. In the standard case, there is a loss of one order. When measuring the error of randomized algorithms, the root-mean-square error is typically used, which, in the unbiased case, is equivalent to the standard deviation:

$$\sqrt{\mathrm{Var}\left(\sum_{j=1}^{n}\int_0^h s^2 ds - h^3 U_j^2\right)} = \sqrt{\mathrm{Var}\left(\sum_{j=1}^{n} h^3 U_j^2\right)} \tag{22}$$

$$= h^3\sqrt{\mathrm{Var}\left(\sum_{j=1}^{n} U_j^2\right)} \tag{23}$$

$$= h^3\sqrt{\sum_{j=1}^{n}\mathrm{Var}(U_j^2)} \tag{24}$$

$$= h^3\sqrt{n\mathrm{Var}(U^2)} \tag{25}$$

$$= h^3\sqrt{n}\sqrt{\mathrm{Var}(U^2)} \tag{26}$$

$$= O(h^{2.5}). \tag{27}$$

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

2.3 Unbiased Non-Linearity

In this subsection, we present techniques for handling non-linearity. It may appear that unbiased approaches are only applicable to linear problems. By utilizing independent samples, it becomes possible to handle polynomial non-linearities. While transforming non-linearities into polynomials may not always be straightforward, it is still possible to develop biased alternative approaches based on linearization or approximate polynomial non-linearities.

Example 2.3.1 ($y' = y^2$)
Consider the following ODE:

$$y' = y^2, \quad y(1) = -1. \tag{28}$$

The solution to this equation is given by $y(t) = -\frac{1}{t}$. By integrating both sides of equation (28), we obtain the following integral equation:

$$y(t) = -1 + \int_1^t y(s)y(s)ds. \tag{29}$$

To estimate the recursive integral in equation (28), we use i.i.d. $Y_1, Y_2 \sim Y$ in following RRVE:

$$Y(t) = -1 + (t-1)Y_1(S)Y_2(S), \tag{30}$$

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

Python Code 2.3.2 (implementation (2.3.1))

```python
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 2.3.3 ($e^{E[X]}$)

$e^{\int x(s)ds}$ 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:

$$e^{E[X]} = \sum_{n=0}^{\infty} \frac{E^n[X]}{n!} \tag{31}$$

$$= 1 + \frac{1}{1}E[X]\left(1 + \frac{1}{2}E[X]\left(1 + \frac{1}{3}E[X]\,(1 + ...)\right)\right). \tag{32}$$

Change the fractions of equation (32) to Bernoulli processes and replace all X with independent X_j with $E[X] = E[X_i]$.

$$e^{E[X]} = E\left[1 + B\left(\frac{1}{1}\right)E[X_1]\left(1 + B\left(\frac{1}{2}\right)E[X_2]\left(1 + B\left(\frac{1}{3}\right)E[X_3]\,(1 + ...)\right)\right)\right] \tag{33}$$

$$= E\left[1 + B\left(\frac{1}{1}\right)X_1\left(1 + B\left(\frac{1}{2}\right)X_2\left(1 + B\left(\frac{1}{3}\right)X_3\,(1 + ...)\right)\right)\right] \tag{34}$$

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

Python Code 2.3.4 (implementation of $e^{E[X]}$ (2.3.3))

The following python code estimates $e^{\int_0^t s^2 ds}$:

```python
from random import random as U
from math import exp
def X(t): return -t**3*U()**2
def num_B(i):  # = depth of Bernoulli's = 1
    return num_B(i+1) if U()*i < 1 else i-1
def res(n, t): return 1 + X(t)*res(n-1, t) if n != 0 else 1
def expE(t): return res(num_B(0), t)

t, nsim = 1, 10**3
sol = sum(expE(t) for _ in range(nsim))/nsim
exact = exp(-t**3/3)
print(f"sol = {sol} %error={(sol- exact)/exact}")
#sol = 0.7075010309320893 %error=-0.01260277046
```

Related Work 2.3.5 (unbiased non-linearity)

A similar approach to non-linearity can be found in [ET19].

2.4 Recursion

In this subsection, we discuss recursion-related techniques.

Technique 2.4.1 (coupled recursion)
The idea behind coupled recursion is sharing recursion calls of multiple RRVEs for simulation. This does make them dependent.

Example 2.4.2 (coupled recursion)
Consider calculating the sensitivity of following ODE to a parameter a:

$$y' = ay, y(0) = 1 \Rightarrow \tag{35}$$
$$\partial_a y' = y + a\partial_a y \tag{36}$$

Turn (35) and (36) into RRVEs. To emphasize that they are coupled, that they should recurse together we write them in a matrix equation:

$$\begin{bmatrix} Y(t) \\ \partial_a Y(t) \end{bmatrix} = X(t) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} a & 0 \\ 1 & a \end{bmatrix} X(Ut). \tag{37}$$

Observe how this eliminates the additive branching recursion present in equation (36).

Python Code 2.4.3 (implementation of (37))

```
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 2.4.4 (coupled recursion)
Example (2.4.2) is inspired by [VSJ21]. [VSJ21] propose an efficient unbiased back-propagation algorithm for rendering. [YVJ22] extends [VSJ21] to walk on spheres.

Technique 2.4.5 (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 2.4.6 (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. When applicable, tail recursion can be used to implement recursion that addresses these concerns.

Technique 2.4.7 (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. However, tail recursion may not always be desirable as it discards intermediate values of the recursion calls which may be of interest. To retain some of these intermediate values while still partly optimizing for performance, it is possible to combine tail recursion with normal recursion.

Python Code 2.4.8 (tail recursion on (37))

We implement (37) 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.

```
from random import random as U
import numpy as np
def X(t, a) -> np.array:
    q, A = np.array([1.0, 0.0]), np.array([[a, 0.0], [1.0, a]])
    sol, W = np.array([1.0, 0.0]), np.identity(2)
    while U() < t:
        W = W @ A if t < 1 else t * W @ A
        sol += W @ q
        t *= U()
    return sol
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) = [2.7198 2.7163]
```

2.5 Limitations and Future Work

There are several MC techniques we did not discuss like quasi-MC and importance sampling. We would like to mention SALT a MC technique that we find particularly interesting and has received relatively little attention.

Technique 2.5.1 (SALT)

Sequential Approximation in L-Two (SALT) is an adaptive MC technique that builds control variates with a (bi-)orthonormal basis. The rough idea behind it is that calculating coefficients for a (bi-)orthonormal basis can be found by MC integration and the current estimate of those coefficients can accelerate MC integration. It can be seen as an interesting case of stochastic gradient descent.

Related Work 2.5.2 (SALT)

SALT gets discussed in [GS14]. Other noteworthy papers on adaptive MC techniques are [He+21] on adaptive important sampling and [Sal+22] on regression-based MC.

Related Work 2.5.3 (MC trapezoidal rule)

In [Wu20], an improvement of half order for a MC trapezoidal rule is discussed. We are uncertain whether this improvement is similar to ours. Further examination would be needed.

Our current method for handling non-linearity, as demonstrated in (2.3.3), is rudimentary and only applicable to small levels of non-linearity. We have not tested any variance reduction techniques in this context. We think that major improvements can be made in (2.3.3).

We did not discuss branching tail recursion but it is important to improve our approach to handling non-linearity. There exist multiple methods of implementing branching tail recursion, each with its own set of benefits and drawbacks.

Technique 2.5.4 (checkpointing)
The structure of branching recursion can be captured by a tree. Storing that tree in memory can be expensive. In recursion, you only need to retrace steps 1 by 1 therefore you only need local parts of the recursion tree. Checkpointing tries to alleviate memory issues by instead storing the whole tree only storing seeds (of the random generator) of parts of the tree and growing them when needed.

Related Work 2.5.5 (branching tail recursion)
Checkpointing appears in [VSJ21].

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:

$$y' = y, \tag{38}$$

but this time with a different condition:

$$\int_0^1 y(s)ds = e - 1. \tag{39}$$

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:

$$G' = \delta(x - t), \quad \int_0^1 G(s, x)ds = 0. \tag{40}$$

Solving this equation yields:

$$G(t, x) = H(t - x) + x - 1. \tag{41}$$

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 (38):

$$y(t) = e - 1 + \int_0^1 G(t,s)y(s)ds. \tag{42}$$

Converting equation (42) into a RRVE using RMC, we obtain:

$$Y(t) = e - 1 + 2B\left(\frac{1}{2}\right) Y(S)(H(t-S) + S - 1), \tag{43}$$

where $S \sim U$. We will skip the Python implementation of equation (43) as it does not provide any new information. Instead, we will plot realizations of equation (43) in Figure 4.

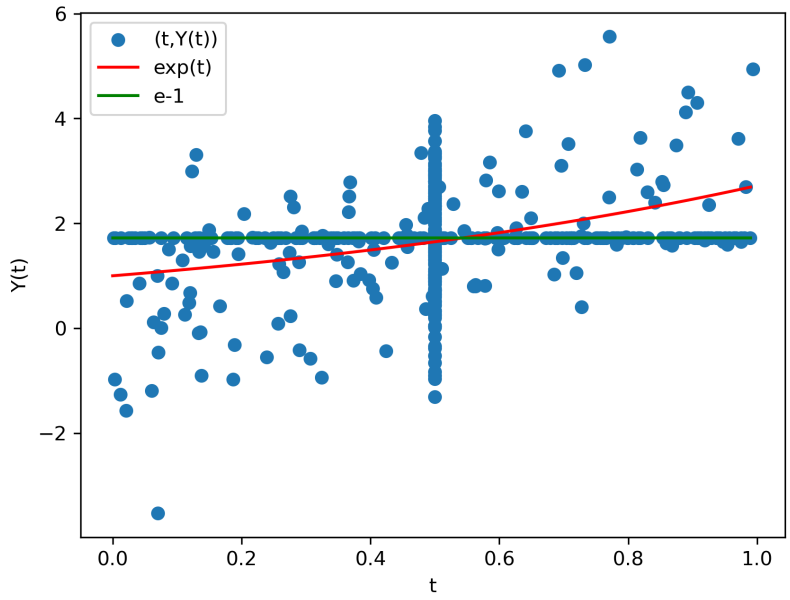

Figure 4: Recursive calls of (43) when calling $Y(0.5)$ 300 times. Points accumulate on the green line due to the Russian roulette, and at $t = 0.5$ because it is the starting value of the simulation.

Example 3.1.2 ($y'' = y$ mixed boundary conditions)
Consider the following ODE:

$$y'' = y, \quad y(0) = 1, \quad y'(1) = e. \tag{44}$$

This equation has the solution $y(t) = e^t$. We define the source green function $G(t,x)$ for y'' and Dirichlet/Neumann boundary conditions as follows:

$$G'' = \delta(t-x), \quad G(0) = 0, \quad G'(1) = 0. \tag{45}$$

Solving this equation yields:

$$G(t,x) = \begin{cases} -t & \text{if } t < x \\ -x & \text{if } t \geq x \end{cases}. \tag{46}$$

We define the boundary green function $P(t,x)$ (for $x \in 0,1$) for y'' and Dirichlet/Neumann boundary conditions as follows:

$$P'' = 0, \quad (P(0,x), P'(1,x)) = \begin{cases} (1,0) & \text{if } x = 0 \\ (0,1) & \text{if } x = 1 \end{cases}. \tag{47}$$

These are a basis for the homogeneous solutions. $P(t,x)$ can be solved from its definition:

$$P(t,x) = \begin{cases} 1 & \text{if } x = 0 \\ t & \text{if } x = 1 \end{cases}. \tag{48}$$

At this point, it may not be clear, but with this set of green functions, we can form the following integral equation for (44):

$$y(t) = P(t,0)y(0) + P(t,1)y(1) + \int_0^1 G(t,s)y(s)ds. \tag{49}$$

Equation (49) can be expressed as:

$$Y(t) = 1 + te + lB\left(\frac{1}{l}\right) G(t,S)Y(S), \tag{50}$$

where $S \sim U$ and $l > 1 \in \mathbb{R}$. We visualize equation (50) in Figure 5.

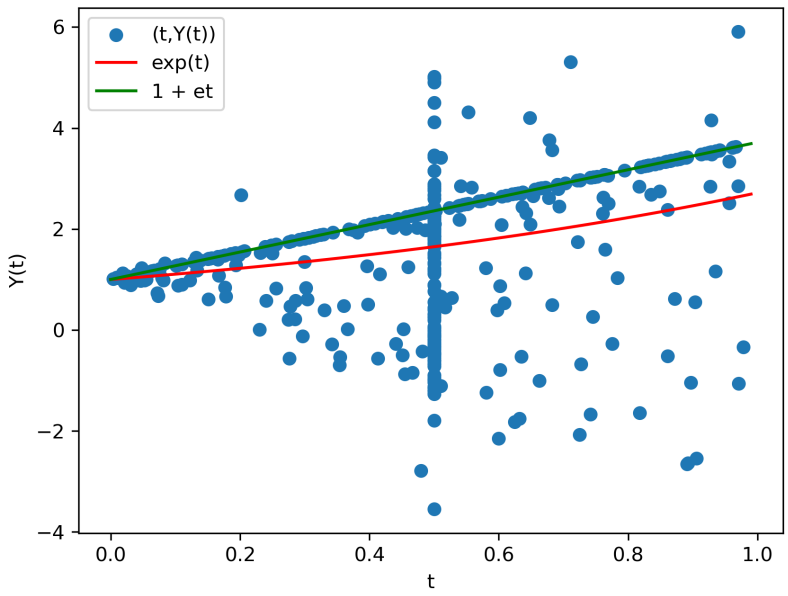

Figure 5: Recursive calls of (50), $l = 2$ when calling $Y(0.5)$ 300 times.

Related Work 3.1.3 ($y'' = y$ mixed boundary conditions)
[Saw+23] discusses an algorithm for mixed boundary conditions.

Definition 3.1.4 (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 3.1.5 (green function)
Our notion of green function is similar to that in [HGM01].

3.2 Fredholm Integral Equations

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:

$$\varphi(t) = f(t) + \lambda \int_a^b K(t,s)\varphi(s)ds. \tag{51}$$

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:

$$y'' = y, \quad y(b_0), y(b_1). \tag{52}$$

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

$$P(t,x) = \begin{cases} \frac{b_1 - t}{b_1 - b_0} & \text{if } x = b_0 \\ \frac{t - b_0}{b_1 - b_0} & \text{if } x = b_1 \end{cases}, \tag{53}$$

$$G(t,s) = \begin{cases} -\frac{(b_1 - t)(s - b_0)}{b_1 - b_0} & \text{if } s < t \\ -\frac{(b_1 - s)(t - b_0)}{b_1 - b_0} & \text{if } t < s \end{cases}. \tag{54}$$

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

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

$$Y(t) = P(t,b_0)y(b_0) + P(t,b_1)y(b_1) + lB\left(\frac{1}{l}\right)(b_1 - b_0)G(t,S)y(S), \tag{56}$$

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 2.1.6), we can also split the domain in Equation (55) as follows:

$$y(t) = P(t, b_0)y(b_0) + P(t, b_1)y(b_1) + \frac{1}{2}\int_{b_0}^{b_1} G(t, s)y(s)ds + \frac{1}{2}\int_{b_0}^{b_1} G(t, s)y(s)ds, \tag{57}$$

$$y(t) = P(t, b_0)y(b_0) + P(t, b_1)y(b_1) + \int_{b_0}^{\frac{b_1+b_0}{2}} G(t, s)y(s)ds + \int_{\frac{b_1+b_0}{2}}^{b_1} G(t, s)y(s)ds. \tag{58}$$

By coupling, we can eliminate the additive branching recursion in the RRVEs corresponding to Equations (57) and (58). This results in the following RRVE:

$$X(t_1, t_2) = \begin{bmatrix} P(t_1, b_0) & P(t_1, b_1) \\ P(t_2, b_0) & P(t_2, b_1) \end{bmatrix} \begin{bmatrix} y(b_0) \\ y(b_1) \end{bmatrix} + W \begin{bmatrix} G(t_1, S_1) & G(t_1, S_2) \\ G(t_2, S_1) & G(t_2, S_2) \end{bmatrix} X(S_1, S_2), \tag{59}$$

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 (59))
We implemented equation (59) 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.

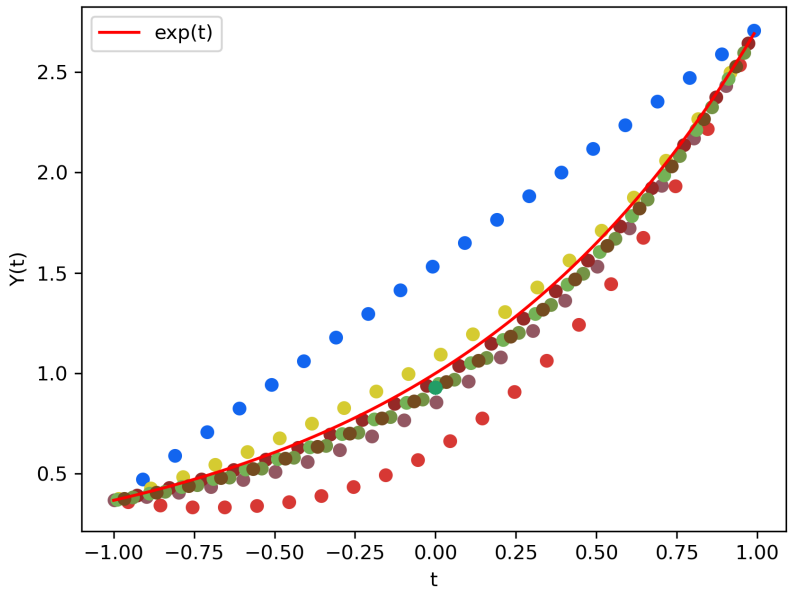

Figure 6: Recursive calls of equation (59) when calling $X(0)$ once, with a split size of 20, S_j are coupled such they are equally spaced and the coupling is colored. The initial conditions for this call are $y(-1) = e^{-1}$ and $y(1) = e^1$, with Russian roulette rate $l = 1.2$.

Related Work 3.2.5 (coupled splitting)
Coupled splitting is partly inspired by how [SM09] reduces variance by using bigger submatrices. Coupled splitting does not work for walk on sphere because of the dynamically changing domain. A technique based on reusing samples such as coupled splitting for walk on sphere gets discussed in [Mil+23].

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

$$y' = y, \quad y(0) = 1. \tag{60}$$

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

$$y(t) = y(t_n) + \int_{t_n}^{t} y(s)ds, \quad t > t_n. \tag{61}$$

Turn these in the following class of RRVEs:

$$Y_n(t) = y(t_n) + (t - t_n)Y_n((t - t_n)U + t_n), \quad t > t_n. \tag{62}$$

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:

$$Y_n(t) = y_n + (t - t_n)Y_n((t - t_n)U + t_n), \quad t > t_n \tag{63}$$

$$y_n = \begin{cases} Y_{n-1}(t_n) & \text{if } n \neq 0 \\ y(t_0) & \text{if } n = 0 \end{cases}. \tag{64}$$

We refer to equation (63) as the inner recursion and equation (64) 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
```

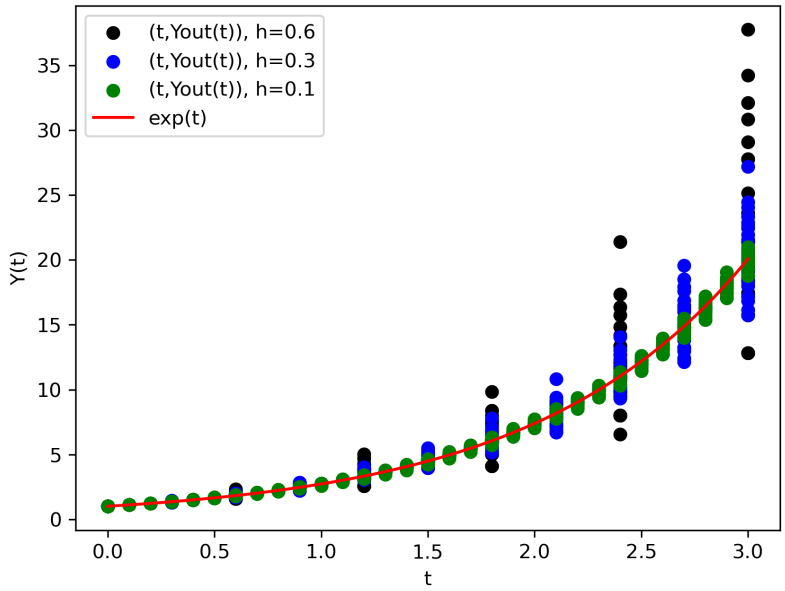

Figure 7: Recursive calls of equation (64) when calling $Y_{out}(3, h)$ 30 times for different h.

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 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 2.2.2).

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

$$y(t) = y(t_n) + \int_{t_n}^{t} y(s)ds, \quad t > t_n. \tag{65}$$

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

$$\begin{aligned} y(s) &= y(t_n) + (s - t_n)y'(t_n) + O((s - t_n)^2) && (66)\\ &\approx y(t_n) + (s - t_n)f(y(t_n), t_n) && (67)\\ &\approx y(t_n) + (s - t_n)\left(\frac{y(t_n) - y(t_{n-1})}{t_n - t_{n-1}}\right) && (68)\\ &\approx y(t_n)(1 + s - t_n). && (69) \end{aligned}$$

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

$$\begin{aligned} y(t) &= y(t_n) + \int_{t_n}^{t} y(s)ds && (70)\\ &= y(t_n) + \int_{t_n}^{t} y(s) - y(t_n)(1 + s - t_n) + y(t_n)(1 + s - t_n)ds && (71)\\ &= y(t_n)\left(1 + (1 - t_n)(t - t_n) + \frac{t^2 - t_n^2}{2}\right) + \int_{t_n}^{t} y(s) - y(t_n)(1 + s - t_n)ds. && (72) \end{aligned}$$

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.

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:

$$y' = y^2 - t^4 + 2t, \quad y(0) = 0. \tag{73}$$

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

$$y(t) = y(t_n) + \int_{t_n}^{t} y^2(s)ds - \frac{t^5 - t_n^5}{5} + (t^2 - t_n^2). \tag{74}$$

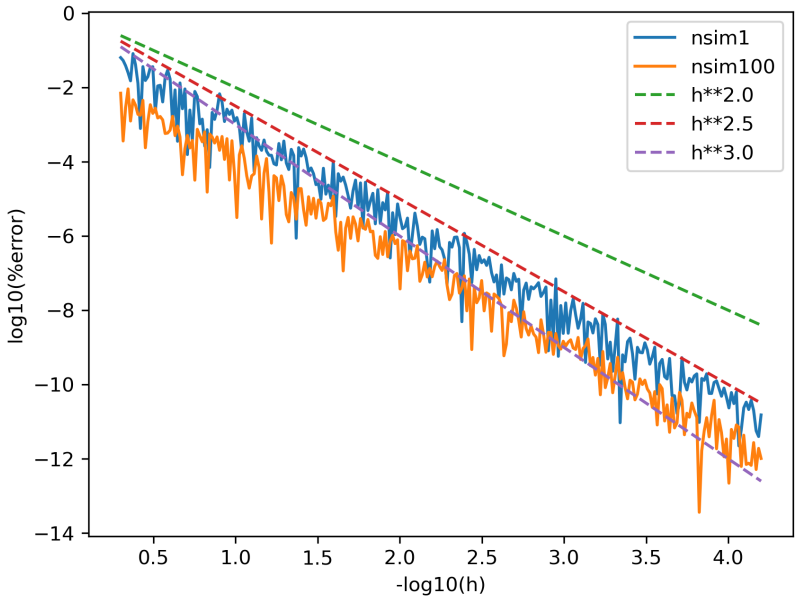

Figure 8: Log-log plot of the error for example (3.3.3) at $Y(10)$.

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

$$y^2(t) \approx y^2(t_n) + 2(t-t_n)y(t_n)y'(t_n) + ((t-t_n)y'(t_n))^2 + O((t-t_n)^2) \tag{75}$$
$$\approx y^2(t_n) + 2(t-t_n)y(t_n)y'(t_n) + O((t-t_n)^2) \tag{76}$$

Integrating the control variate yields:

$$\int_{t_n}^{t} y^2(t_n) + 2(s-t_n)y(t_n)y'(t_n)ds \tag{77}$$
$$= (t-t_n)y^2(t_n) + 2\left(\frac{t^2-t_n^2}{2} - t_n(t-t_n)\right)y(t_n)y'(t_n). \tag{78}$$

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, 1)
        tn = tt
    return yn
```

3.4 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 9 illustrates that arbitrary RMC do not converge.

Coupled splitting was tested on the example shown in Figure 9, but it did not contribute to the convergence of the method. This suggests that a fix point argument would not be effective for this particular example with this Russian roulette setup.

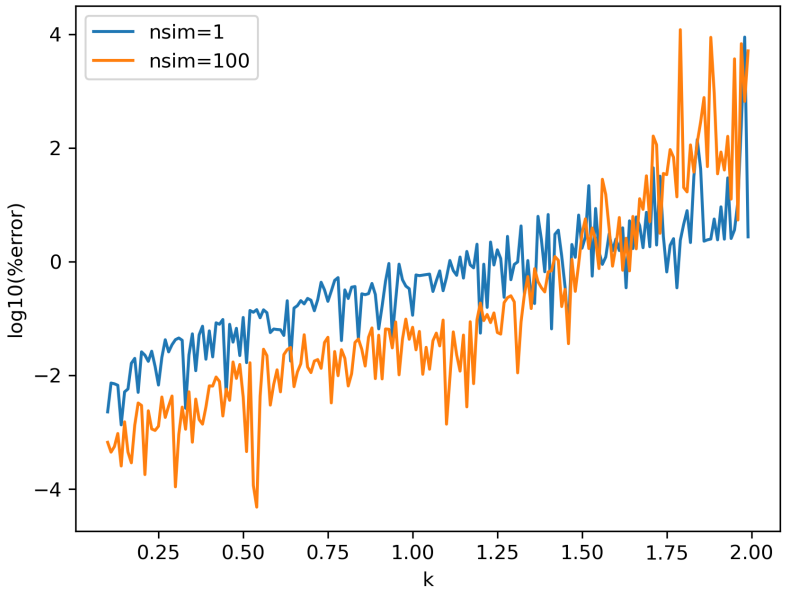

Figure 9: The logarithmic percentage error of $Y(0)$ for (56), with $l = 1.2$ and initial conditions $y(-k) = e^{-k}$ and $y(k) = e^{k}$, displays an exponential increase until approximately $k = 1.5$, beyond which additional simulations fail to reduce the error, indicating that the variance doesn't exist.

Figure 6 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 in-

creasing coupled splitting amount when approaching the bottleneck. Alternatively when reaching the bottleneck it is possible to rely on MC convergence.

Related Work 3.4.1 (convergence coupled splitting)

See [GH21] for a discussion on the convergence of recursive stochastic algorithms. We highly recommend watching the corresponding video [Abh20] before reading the paper.

Similarly to classic methods, RRMC struggles with big negative coefficients in front of the recursive parts which may appear in stiff problems. We tested DRRMC as a potential remedy but we conclude it to be ineffective.

Definition 3.4.2 (DRRMC)

Consider a general linear ODE IVP problem:

$$x' = Ax + g, \quad x(0) = x_0. \tag{79}$$

Sometimes repeatedly multiplying by A is unstable. Diagonal RRMC adds a positive diagonal matrix D to A and hopes that it stabilizes.

$$x' + Dx = (A + D)x + g. \tag{80}$$

The following integral equation can be derived by using integrating factor:

$$x(t) = e^{D(t_n - t)}x(t_n) + \int_{t_n}^{t} e^{D(s-t)}(A + D)x(s)ds + \int_{t_n}^{t} e^{D(s-t)}g(s)ds. \tag{81}$$

Remember that the exponential of a diagonal matrix is the exponential of its elements. The recursive integral has the following trivial control variate:

$$\int_{t_n}^{t} e^{D(s-t)}(A + D)x(t_n)ds = D^{-1}(I - e^{D(t_n - t)})(A + D)x(t_n). \tag{82}$$

Note that D may be chosen differently for every outer recursion.

Example 3.4.3 (DRRMC)

Consider:

$$x' = Ax, x(0) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}. \tag{83}$$

With

$$A = \begin{bmatrix} 0 & 1 \\ -1000 & -1001 \end{bmatrix}. \tag{84}$$

This has the following solution:

$$x(t) = \frac{1}{999}\begin{bmatrix} -e^{-1000t} + 1000e^{-t} \\ 1000e^{-1000t} - 1000e^{-t} \end{bmatrix}. \tag{85}$$

We choose D fixed over all outer recursions:

$$D = \begin{bmatrix} 1 & 0 \\ 0 & 1000 \end{bmatrix}. \tag{86}$$

We make the convergence plot for this example with integral equation (81) with control variate (82) implemented with recursion in recursion on Figure 10.

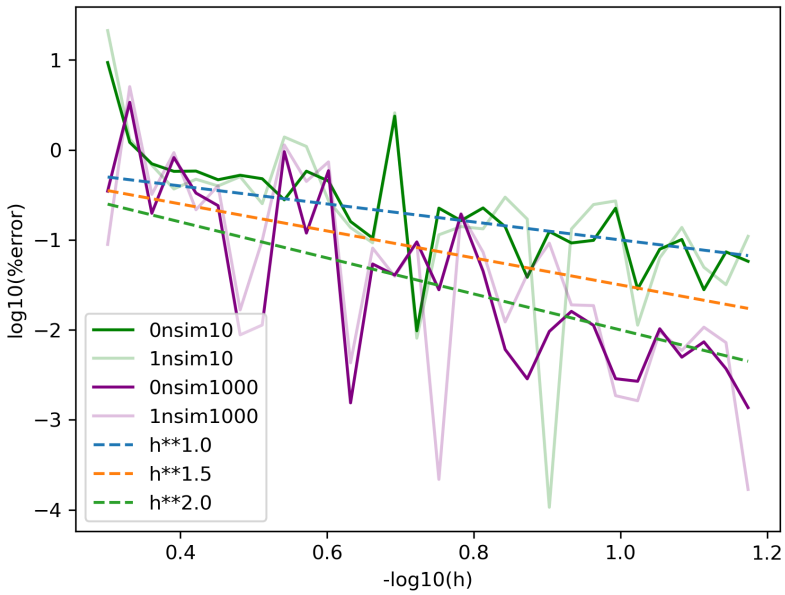

Figure 10: Log-log plot of the error of example (3.4.3). We plotted the second component of the error transparent.

Related Work 3.4.4 (DRRMC)

DRRMC is inspired by $\bar{\sigma}$ parameter in [Saw+22] but instead of importance sampling we use control variates to deal with nonlinearity introduced by the exponential because it needs to work over an entire vector at the same time. Ideas from exponential integrator methods may improve DRRMC.

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 3.4.5 (random ODEs)

Consider the problem given by the initial value problem

$$Y' = AY, \quad Y(0) = 1, \quad A \sim \text{Uniform}(0, 1). \tag{87}$$

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:

$$E_A[f(Y(t))] = E_A[f(Y(t)) \mid A] \tag{88}$$

$$= E_A[f(E_X[X(t, A)]) \mid A]. \tag{89}$$

To estimate $f(E_X[X(t,A)])$, we use the approach outlined in (2.3.3). For the specific example of (87), we can compute the first two moments of $Y(t)$ as

$$E_A[Y(t)] = \frac{e^t}{t} - \frac{1}{t}, \tag{90}$$

$$E_A[Y^2(t)] = \frac{e^{2t}}{2t} - \frac{1}{2t}. \tag{91}$$

Python Code 3.4.6 (implementation of (3.4.5))

```
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 Brownian Motion

Current RMC algorithms for PDEs are related to Brownian motion. In this section, we explore the relationship between the heat equation and Brownian motion, and discuss how recursive first-passage sampling fits into the picture.

4.1 Heat Equation

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

Lemma 4.1.1 (self-affinity Brownian motion)
Brownian motion is a self-affine random process, which implies that any subpath can be translated and scaled in such a way that its distribution matches that of the entire path.

$$\forall c \in \mathbb{R}_0^+ : \frac{W_{ct}}{\sqrt{c}} \sim W_t. \tag{92}$$

Definition 4.1.2 (1D heat equation Dirichlet)

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

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}. \tag{93}$$

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

Figure 11 shows how the Dirichlet condition is defined for parabola but reverse in time.

Lemma 4.1.3 (Brownian motion and the heat equation)
For problem (4.1.2) if $|\psi|$ is bounded then there holds:

$$u(x,t) = E[\psi(Y_\tau, \tau)|Y_t = x]. \tag{94}$$

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:

$$\frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}. \tag{95}$$

Isolate $u(x,t)$:

$$u(x,t) = \frac{\Delta t}{2\Delta t + \Delta x^2}\left(u(x+\Delta x,t) + u(x-\Delta x,t)\right) + \frac{\Delta x^2}{2\Delta t + \Delta x^2}\left(u(x,t-\Delta t)\right). \tag{96}$$

Because $u(x+\Delta x,t) \approx u(x-\Delta x,t) \approx u(x,t-\Delta t) \approx$ right-hand side of equation (96) we may Russian roulette to remove branching recursion and generate a recursion path instead of a tree.

$$Z(x,t) = \begin{cases} \psi(\text{argmin}_{b\in\partial\Omega}||(x,t)-b||) & \text{when } (x,t) \notin \Omega \\ \begin{cases} Z(x+\Delta x,t) & \text{with chance } \frac{\Delta t}{2\Delta t+\Delta x^2} \\ Z(x-\Delta x,t) & \text{with chance } \frac{\Delta t}{2\Delta t+\Delta x^2} \\ Z(x,t-\Delta t) & \text{with chance } \frac{\Delta x^2}{2\Delta t+\Delta x^2} \end{cases} & \text{else .} \end{cases} \tag{97}$$

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 (97) the limit makes the recursion path go to Brownian motion Y_t.

$$Z(x,t) \to \psi(Y_\tau, \tau). \tag{98}$$

Finishing the proof. □

Related Work 4.1.4

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

4.2 First Passage Sampling

In this subsection we build up to sampling (Y_τ, τ) from (4.1.3) efficiently and extend this to paths.

Definition 4.2.1 (first passage time)
Define the first passage time for a process X_t for a set of valid states V as

$$\mathrm{FPt}(X_t, V) = \inf\{t > 0 | (X_t, t) \notin V\}. \tag{99}$$

Note that the first passage time is a RV itself.

Definition 4.2.2 (first passage)
Define the first passage for a process X_t for a set of valid states V as

$$\mathrm{FP}(X_t, V) = (X_\tau, \tau), \tau = \mathrm{FPt}(X_t, V). \tag{100}$$

Theorem 4.2.3
The the density of first passages of Brownian motion for exiting a boundary is the Dirichlet boundary green function for the heat equation.

Proof. Follows from (4.1.3). □

Example 4.2.4 (Euler first passage sampling)
In this example, we approximately sample the first passage of Brownian motion for a parabolic barrier by simulating Brownian motion with the Euler scheme. We plotted this in Figure 11.

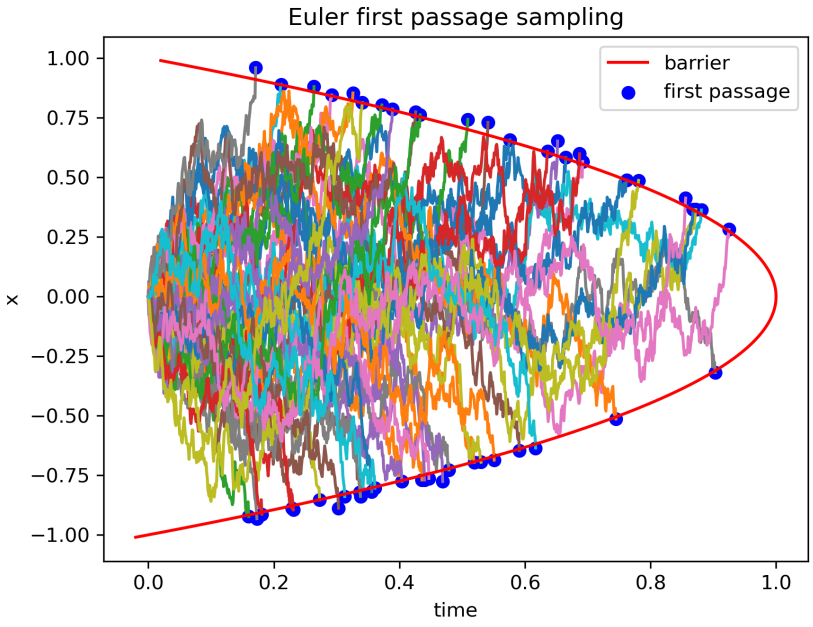

Figure 11: 50 *realizations of Euler first passage sampling with step size* 0.001.

Lemma 4.2.5

When a process has more valid states the first passage time gets larger i.e.

$$V_1 \subset V_2 \Rightarrow \mathrm{FPt}(X_t(\omega), V_1) \leq \mathrm{FPt}(X_t(\omega), V_2). \tag{101}$$

The ω is to indicate we mean the same realization of X_t.

Technique 4.2.6 (recursive first passage sampling)

Recursive first passage sampling involves sampling an initial, simpler first passage, the base that includes fewer valid states. Using this sampled first passage as a starting point, we then perform the same sampling process until the sampled first passage is almost invalid.

Example 4.2.7 (recursive first passage sampling)

In this example, we sample the first passage of Brownian motion from a parabolic barrier with recursive first passage sampling. For the simpler first passage sampler, we scale and translate samples from a triangular barrier so its valid states are contained in the parabola generated by the Euler scheme. The precomputed samples are created by (4.2.8) and used to produce first passages in (4.2.9).

Python Code 4.2.8 (Euler first passage sampling)

```
import numpy as np
def in_triangle(time, pos): return 1-time > abs(pos)
def sample_euler_triangle(dt=0.001):
    pos, time = 0, 0
    while in_triangle(time, pos):
        pos += np.random.normal(0, 1) * np.sqrt(dt)
        time += dt
    return (time, pos)
```

Python Code 4.2.9 (recursive first passage sampling)

The maximum scaling of the triangular barrier that fits in the parabola is derived through (4.1.1) and using the fact that a parabola domain is convex. To dampen barrier overstepping of (4.2.8) we use a smaller scaling than the maximum.

```
import numpy as np
import random
def triangle_scale_in_para(time, pos):
    xx = np.sqrt(1-time) - abs(pos)
    tt = abs(1-abs(pos)**2-time)
    return np.sqrt(tt) if np.sqrt(tt) < xx else xx
# requires precomputed triangle_sample
def sample_recursive_para(accuracy=0.01, scale_mul=0.9):
    time, pos = 0, 0
    scale = triangle_scale_in_para(time, pos)
    while scale > accuracy:
        scale *= scale_mul
        dtime, dpos = random.sample(triangle_sample, 1)[0]
        dtime, dpos = (scale**2)*dtime, scale*dpos
        pos += dpos
        time += dtime
        scale = triangle_scale_in_para(time, pos)
    return (time, pos)
```

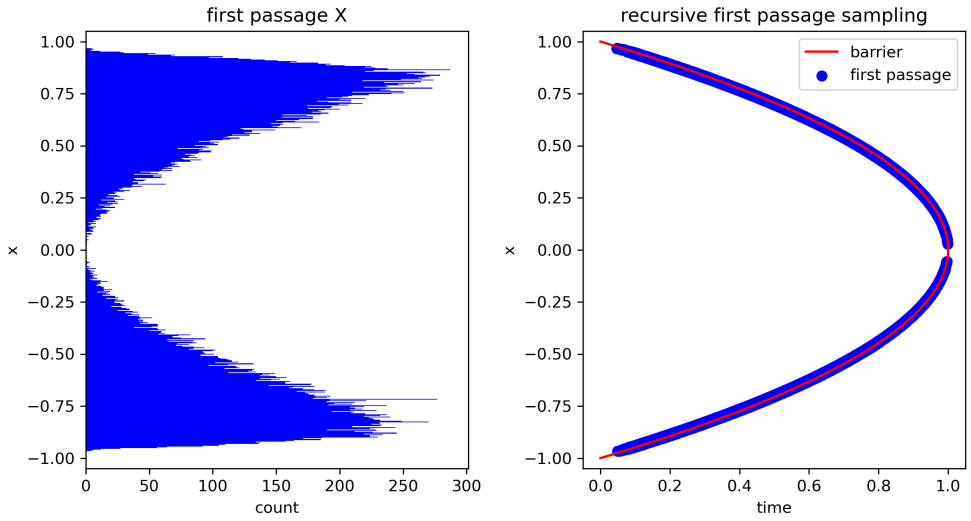

Figure 12: 50000 *of realizations of recursive first passage sampling produced by (4.2.9). The precomputed sample of* 5000 *first passages of a triangular barrier uses the Euler scheme with step size* 0.001.

Related Work 4.2.10 (recursive first passage sampling)

The original walk on spheres is a recursive first passage algorithm. Recursive first passage sampling for Brownian motion is discussed in [HT16] and by transformation also first passage problems for the Ornstein-Uhlenbeck process. An alternative to resampling from an Euler scheme is to use tabulated inverse cumulative probability functions, as demonstrated in [HGM01].

Example 4.2.11 (recursive first passage average sampling)

In example (4.2.7) it is possible to keep track of the average because it scales and translates with our base first passage sampler.

Technique 4.2.12 (Brownian motion path stitching)

Instead of thinking of sampling the first passages you can also sample whole paths to the first passage. Similar to before we need to be able to generate paths for a simple first passage problem and "stitch" these paths together. An advantage over normally generating paths is that a path can be represented by its subpaths and their scalings requiring less memory than a fully stored path. This can be useful in case you need to look back to a part of the path. The only downsides are that the time steps are inhomogeneous and it requires storing paths for the simpler first passage problem.

Example 4.2.13 (path stitching parabola)

This is the same as example (4.2.7) but now we have to keep track of the whole resampled Euler scheme generated paths.

Related Work 4.2.14 (path stitching)

Path stitching appears frequently in rendering and also in [Das+15] and [JL12].

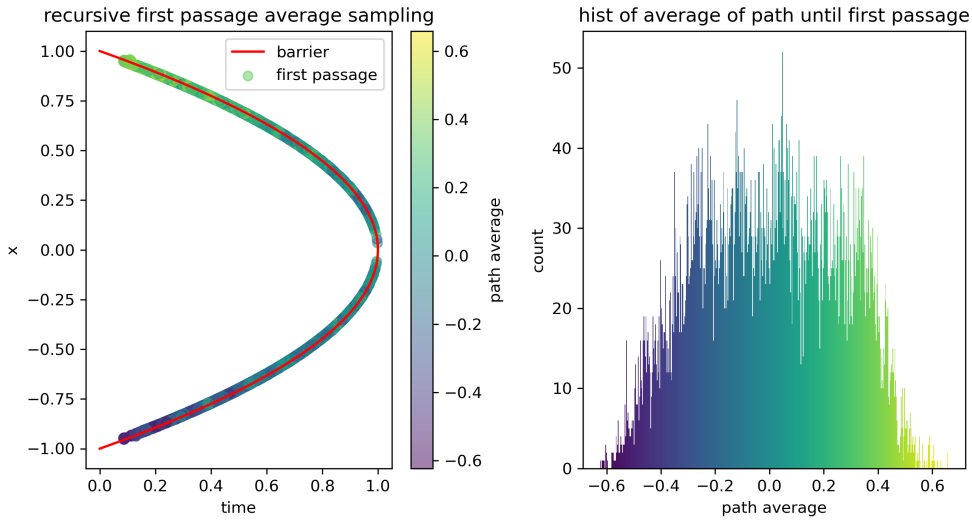

Figure 13: 10000 *of realizations of recursive first passage sampling (for the average). The precomputed sample of* 1000 *first passages and averages of a triangular barrier uses the Euler scheme with step size* 0.001.

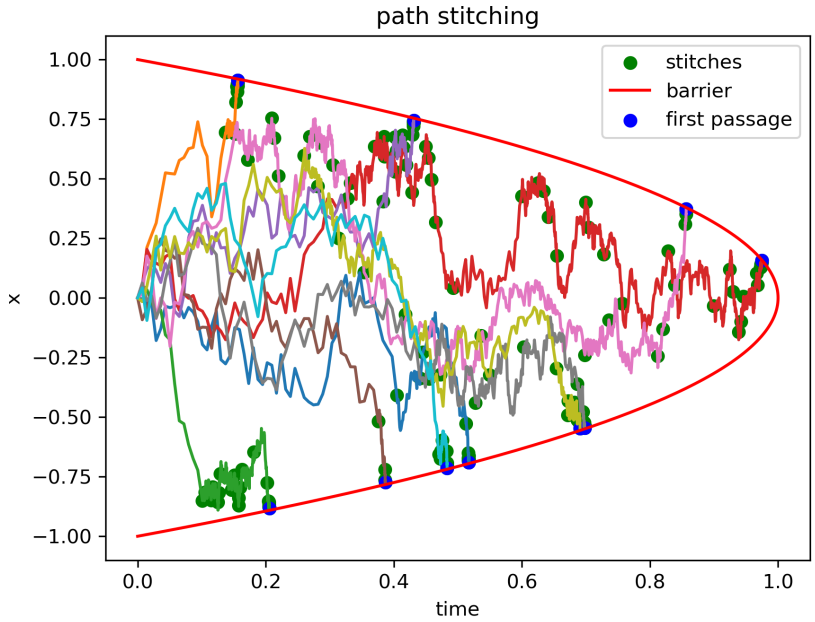

Figure 14: 10 *paths build with path stitching build out of precomputed Euler scheme generated paths with step size* 0.01.

4.3 Limitations and Future Work

It would be interesting to extend proof technique of (4.1.3) to the full Feynman-Kac formula. Just adding a source term f in the Feynman-Kac formula adds an integral of the source term over the recursion path. In equation (96) it would add $\frac{\Delta t \Delta x^2}{2\Delta t + \Delta x^2} f(x,t)$ term. This term can either be kept or Russian rouletted to prevent

infinite source evaluations in the limit. This roughly corresponds to estimating the integral by MC integration also requires knowing a finite amount of intermediate points of the recursion path sampled randomly. These can possibly be obtained from the intermediate first passages of recursive first passage sampling or with a smart application of path stitching.

Recursive first passage sampling for Brownian motion can be extended to geometric Brownian motion by transforming the space. However, this transformation breaks the symmetries required for the average used in the example (4.2.11). An unsatisfying way to fix this is to approximate the base sampler at different points in space and at different scales. Just accelerating a classic Euler scheme at critical places with a base sampler can reduce precomputation requirements.

Abstract

Deze scriptie onderzoekt unbiased Monte Carlo methoden 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

[Abh20] Abhishek Gupta. *Recursive Stochastic Algorithms: A Markov Chain Perspective*. Oct. 2020. URL: https://www.youtube.com/watch?v=f1IP6rpqaEE (visited on 01/11/2023).

[Das+15] Atish Das Sarma et al. "Fast distributed PageRank computation". en. In: *Theoretical Computer Science* 561 (Jan. 2015), pp. 113–121. ISSN: 03043975. DOI: 10.1016/j.tcs.2014.04.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S0304397514002709 (visited on 01/05/2023).

[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).

[ET19] S. M. Ermakov and T. M. Tovstik. "Monte Carlo Method for Solving ODE Systems". en. In: *Vestnik St. Petersburg University, Mathematics* 52.3 (July 2019), pp. 272–280. ISSN: 1063-4541, 1934-7855. DOI: 10.1134/S1063454119030087. URL: https://link.springer.com/10.1134/S1063454119030087 (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).

[GS14] Emmanuel Gobet and Khushboo Surana. "A new sequential algorithm for L2-approximation and application to Monte-Carlo integration". en. In: (2014).

[He+21] Shengyi He et al. *Adaptive Importance Sampling for Efficient Stochastic Root Finding and Quantile Estimation*. en. arXiv:2102.10631 [math, stat]. Feb. 2021. URL: http://arxiv.org/abs/2102.10631 (visited on 02/10/2023).

[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).

[HT16] Samuel Herrmann and Etienne Tanré. "The first-passage time of the Brownian motion to a curved boundary: an algorithmic approach". en. In: *SIAM Journal on Scientific Computing* 38.1 (Jan. 2016). arXiv:1501.07060 [math], A196–A215. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/151006172. URL: http://arxiv.org/abs/1501.07060 (visited on 12/20/2022).

[JL12] Hao Ji and Yaohang Li. "Reusing Random Walks in Monte Carlo Methods for Linear Systems". en. In: *Procedia Computer Science* 9 (2012), pp. 383–392. ISSN: 18770509. DOI: 10.1016/j.procs.2012.04.041. URL: https://linkinghub.elsevier.com/retrieve/pii/S1877050912001627 (visited on 09/17/2022).

[JN09] A. Jentzen and A. Neuenkirch. "A random Euler scheme for Carathéodory differential equations". en. In: *Journal of Computational and Applied Mathematics* 224.1 (Feb. 2009), pp. 346–359. ISSN: 03770427. DOI: 10.1016/j.cam.2008.05.060. URL: https://linkinghub.elsevier.com/retrieve/pii/S0377042708002136 (visited on 03/06/2023).

[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).

[Rat+22] Alexander Rath et al. "EARS: efficiency-aware russian roulette and splitting". en. In: *ACM Transactions on Graphics* 41.4 (July 2022), pp. 1–14. ISSN: 0730-0301, 1557-7368. DOI: 10.1145/3528223.3530168. URL: https://dl.acm.org/doi/10.1145/3528223.3530168 (visited on 02/10/2023).

[Sal+22] Corentin Salaün et al. *Regression-based Monte Carlo Integration*. en. arXiv:2211.07422 [cs]. Nov. 2022. URL: http://arxiv.org/abs/2211.07422 (visited on 01/05/2023).

[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).

[Saw+23] Rohan Sawhney et al. *Walk on Stars: A Grid-Free Monte Carlo Method for PDEs with Neumann Boundary Conditions*. en. arXiv:2302.11815 [cs]. Feb. 2023. URL: http://arxiv.org/abs/2302.11815 (visited on 05/01/2023).

[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).

[Wu20] Yue Wu. *A randomised trapezoidal quadrature*. en. arXiv:2011.15086 [cs, math]. Dec. 2020. URL: http://arxiv.org/abs/2011.15086 (visited on 03/06/2023).

[YVJ22] Ekrem Fatih Yılmazer, Delio Vicini, and Wenzel Jakob. *Solving Inverse PDE Problems using Grid-Free Monte Carlo Estimators*. en. arXiv:2208.02114 [cs, math]. Aug. 2022. URL: http://arxiv.org/abs/2208.02114 (visited on 12/13/2022).