Universiteit
Antwerpen

Masterproef scriptie

Recursive Monte Carlo

Auteur: *Isidoor Pinillo Esquivel*

Promotor: *Wim Vanroose*

Academiejaar 2022-2023

Contents

Abstract

We will write this at the end. Also need a dutch abstract

1 Introduction

1.1 Introductory Example

In this subsection we introduce recursive Monte Carlo with our main example for initial value problems:

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

Integrating both sides of (1) obtains:

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

Equation (2) is a recursive integral equation or to be more specific a linear Volterra integral equation of the second type. By estimating the recursive integral of equation (2) with Monte Carlo one derives following estimator:

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

With $U \sim \text{Uniform}(0, 1)$. If y is well behaved then $E[Y(t)] = y(t)$ but we can't calculate $Y(t)$ without accesses to $y(s), s < t$. Notice that we can replace y by a unbiased estimator of it without changing $E[Y(t)] = y(t)$ by the law of total expectance ($E[X] = E[E[X|Z]]$). By replacing y by 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 you would implement equation (4) with recursion it will run indefinitely. A biased way of around this is by approximating $Y(t) \approx 1$ near $t = 0$. Later we discuss Russian roulette (2.1.1) which can be used as an unbiased stopping mechanism.

Python Code 1.1.1 (implementation of (4))

```
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 equation, it can be helpful to plot all recursive calls $(t, Y(t))$, as shown in Figure 1 for this implementation.

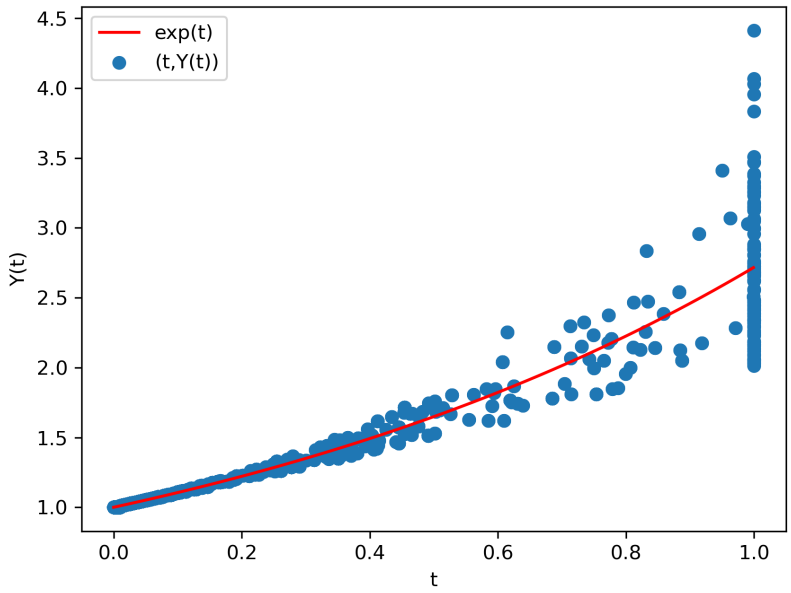

Figure 1: Recursive calls of (1.1.1)

An issue with (1.1.1) is that the variance increases rapidly when t increases. Later this is issue gets resolved in (3.3.1). Note that (1.1.1) keeps desirable properties from unbiased Monte Carlo methods such as being embarrassingly parallel and having simple error estimates.

1.2 Notation

Notations used in this thesis include:

- $B(p) \sim \text{Bernoulli}(p) = \begin{cases} 1 & \text{if } U < p \\ 0 & \text{else} \end{cases}$.
- $H(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{else} \end{cases}$.
- $U \sim \text{Uniform}(0, 1)$.
- BVP = Boundary Value Problem.
- FP = First Passage.
- 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 its self.
- RV = Random Variable.
- Random variables will be denoted with capital letters, e.g., X, Y or Z.

1.3 Related Work

The main motivation for this thesis is the walk on sphere method for 2nd order elliptic PDEs with varying coefficients with Dirichlet boundary conditions discussed in Sawhney et al.'s 2022 [Saw+22] paper. This recursive Monte Carlo method is accurate even if the boundary conditions are geometric complex. We studied the mechanics behind these type of 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 higher order of convergence order optimal even then deterministic algorithms under the same smoothness conditions 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 for nonlinear based on the 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 contributions are:

- An unbiased Monte Carlo method (3.3.3) for linear initial value problems for systems of ordinary differential equations. This method achieves a higher order of convergence in the time step size that is comparable to explicit deterministic methods.
- Coupled recursion (2.4.2) and recursion in recursion (2.4.5) reformulations of the coupling in [VSJ21] and next flight in rendering see [Saw+22] the next flight version of walk on sphere.

In the background section we introduce Monte Carlo techniques, a Monte Carlo trapezoidal rule and discuss coupled recursion / recursion in recursion.

In the ordinary differential equation section we introduce green functions to transform ODEs to integral equations and build up to the our main algorithm for linear initial value problems for systems of ordinary differential equations(3.3.3).

In the Brownian motion section we derive the connection of Brownian Motion to the heat equation with recursive Monte Carlo and 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 used in rendering. The main idea behind Russian roulette is to replace a random variable with a less computationally expensive approximation sometimes.

Definition 2.1.1 (Russian roulette)
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 the following way:

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

Example 2.1.2 (Russian roulette)
Say that we are interested in estimating $E[Z]$ with Z defined in the following way:

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

where $f : \mathbb{R} \to [0, 1]$ expensive to compute. Estimating $E[Z]$ directly would require calling f each simulation. We can modify Z to

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

Now $\tilde{Z}$ just requires calling f on average once every 100 simulations with the variance only increasing slightly compared to Z.

Related Work 2.1.3 (Russian roulette)
In 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 asymptotical 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 in a renderer.

Example 2.1.4 (Russian roulette on (4))
Russian roulette can fix the indefinite recursion issue of equation (4) by approximating Y near $t = 0$ with 1 sometimes. Concretely we replace the t in front of the recursive term with $B(t)$ when $t < 1$.

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

Python Code 2.1.5 (Russian roulette on (4))

```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, $Y(t)$ is constrained to take on only integer values. This is visually evident on Figure 2.

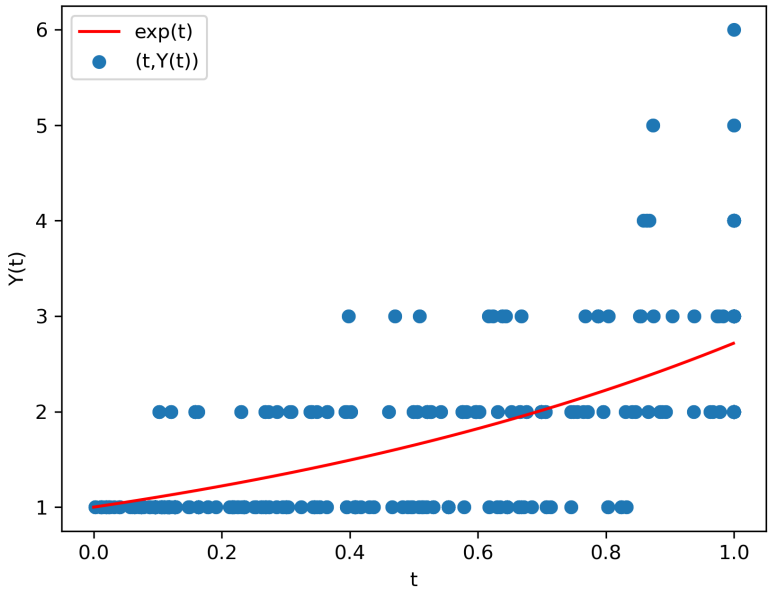

Figure 2: Recursive calls $(t, Y(t))$ of (2.1.5)

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 samples) which reduces the variance.

Definition 2.1.6 (splitting)
Splitting X means using multiple $X_j \sim X$ not independent per se to lower variance by averaging them:

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

Splitting the recursive term in a RRVE can lead to (additive) branching recursion, which requires extra care to ensure that the branches get terminated quickly to avoid an exponential increase in computational complexity. This can be achieved by employing termination strategies that have already been discussed. Later on, we will discuss the use of coupled recursion as a technique for alleviating additive branching recursion in RRVEs (see (3.2.3)).

Example 2.1.7 (splitting on (4))
We can "split" the recursive term of (4) in 2:

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

with $Y_1(t), Y_2(t)$ i.i.d. $Y(t)$.

Python Code 2.1.8 (splitting on (4))

```python
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 (2-level MC)
2-level MC on X with parameters $\tilde{X}, Y : E[\tilde{X}] = E[Y]$:

$$X \to X - \tilde{X} + Y. \tag{11}$$

Definition 2.1.10 (control variates)
Control variate on $f(X)$ is

$$f(X) \to f(X) - \tilde{f}(X) + E[\tilde{f}(X)]. \tag{12}$$

Control variates are a special case of 2-level MC. Usually $\tilde{f}$ is an approximation of f to reduce variance.

Example 2.1.11 (control variate on (4))
To make a control variate for (4) that reduces variance we use following approximation of $y(t) \approx 1 + t$:

$$Y(t) = 1 + t + \frac{t^2}{2} + t(Y(Ut) - 1 - Ut). \tag{13}$$

Notice that we can cancel the constant term of the control variate but that would affect the Russian roulette negatively.

Python Code 2.1.12 (control variate on (4))

```
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 2.1.13 (MC modification)
Our favorite work that discusses these techniques is [Vea]. More interesting MC techniques can be found in rendering. 2-level gets discussed in [Gil13] a introductory paper to multilevel MC.

2.2 Monte Carlo Trapezoidal Rule

We present in this subsection a MC trapezoidal rule with similar convergence behavior to methods discussed later. The MC trapezoidal rule will just be regular MC control variated with the normal trapezoidal rule.

Definition 2.2.1 (MC trapezoidal rule)
Define the MC trapezoidal rule for f on $[x, x+dx]$ the following way:

$$\int_x^{x+dx} f(s)ds \approx \frac{f(x)+f(x+dx)}{2} + f(S_x) - f(x) - \frac{S_x - x}{dx}\left(f(x+dx) - f(x)\right) \quad (14)$$

with $S_x = \text{Uniform}(x, x+dx)$.

Defining the composite MC trapezoidal rule as the sum of MC trapezoidal rules on equally divided intervals is possible but expensive. Every interval would add a function call compared to the normal composite MC trapezoidal rule. Instead you can aggressively Russian roulette into the normal trapezoidal rule such that the increase in functions calls is arbitrarily small.

Definition 2.2.2 (composite MC trapezoidal rule)
Define the composite MC trapezoidal rule for f on $[a, b]$ with n intervals and a Russian roulette rate l the following way:

$$\int_a^b f(s)ds \approx \quad (15)$$

$$\sum_x \frac{f(x)+f(x+dx)}{2} + lB\left(\frac{1}{l}\right)\left(f(S_x) - f(x) - \frac{S_x - x}{dx}(f(x+dx) - f(x))\right) \quad (16)$$

with $S_x = \text{Uniform}(x, x+dx)$.

Python Code 2.2.3 (implementation of (2.2.2))
We implement (2.2.2) for $\int_0^1 e^s ds$.

```python
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(x)+f(x+1/n))/2
    for x in np.arange(0, 1, 1/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:8.794793540941216e-11
```

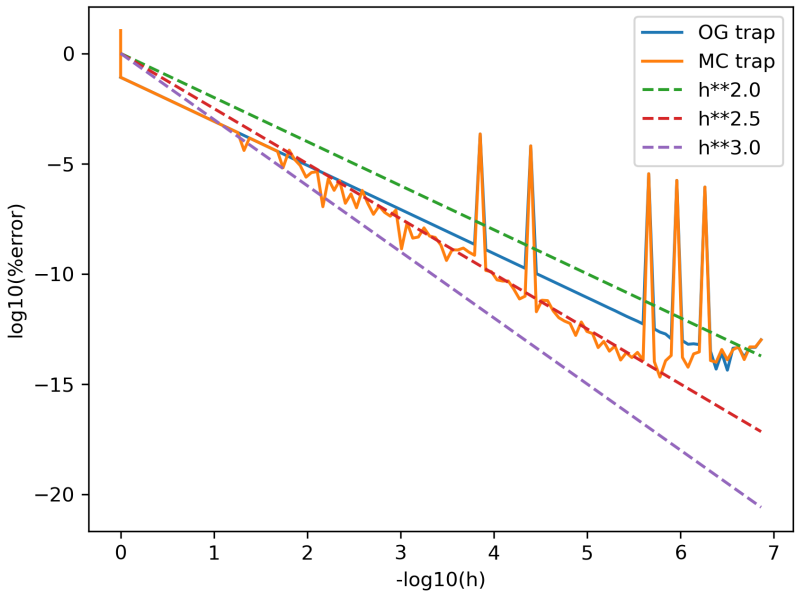

Figure 3: Log-log plot of the error of (2.2.2) for $\int_0^1 e^s ds$ with $l = 100$. The spikes in the graph are artifacts of floating-point arithmetic.

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:

$$\int_0^h s^2 ds = O(h^3). \tag{17}$$

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 like:

$$\int_0^h s^2 ds - h(hU)^2 = \int_0^h s^2 ds - h^3 U^2 = O(h^3). \tag{18}$$

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:

$$\sqrt{\text{Var}\left(\sum_{j=1}^{n} \int_0^h s^2 ds - h^3 U_j^2\right)} = \sqrt{\text{Var}\left(\sum_{j=1}^{n} h^3 U_j^2\right)} \tag{19}$$

$$= h^3 \sqrt{\text{Var}\left(\sum_{j=1}^{n} U_j^2\right)} \tag{20}$$

$$= h^3 \sqrt{\sum_{j=1}^{n} \text{Var}(U_j^2)} \tag{21}$$

$$= h^3 \sqrt{n \text{Var}(U^2)} \tag{22}$$

$$= h^3 \sqrt{n} \sqrt{\text{Var}(U^2)} \tag{23}$$

$$= O(h^{2.5}). \tag{24}$$

Related Work 2.2.4 (MC trapezoidal rule)
In [Wu20] an improvement of a half order for a MC trapezoidal is also discussed but we aren't sure that it is similar to our half order improvement.

2.3 Unbiased Non-Linearity

In this subsection we introduce techniques to deal with non-linearity. At first it may looks only possible to deal with linear problems in an unbiased way but by using independent samples it possible to deal with polynomial non-linearity's (which theoretically extend to any continuos functions by the Weierstrass approximation theorem). It is not always easy to transform non-linearity into polynomials but it is not difficult to come up with biased alternative approaches based on linearization or approximate polynomial non-linearity.

Example 2.3.1 ($y' = y^2$)
Let's do following example:

$$y' = y^2, y(1) = -1. \tag{25}$$

This has solution $-\frac{1}{t}$. Integrate both sides of equation (25) to arrive at following integral equation:

$$y(t) = -1 + \int_1^t y(s)y(s)ds. \tag{26}$$

To estimate the recursive integral in equation (2) we use 2 independent $Y_1, Y_2 \sim Y$:

$$Y(t) = -1 + (t-1)Y_1(S)Y_2(S). \tag{27}$$

With $S \sim \text{Uniform}(1,t)$. This is a branching RRVE this is typical when dealing with non-linearity.

Python Code 2.3.2 ($y' = y^2$)

```
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
```

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

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

$$e^{E[X]} = \sum_{n=0}^{\infty} \frac{E^n[X]}{n!} \tag{28}$$

$$= 1 + \frac{1}{1}E[X]\left(1 + \frac{1}{2}E[X]\left(1 + \frac{1}{3}E[X]\,(1 + ...)\right)\right). \tag{29}$$

Change the fractions of equation (29) to Bernoulli processes and replace all X with independent X_j with $E[X] = E[X_i]$.

$$e^{E[X]} = E\left[1 + B\left(\frac{1}{1}\right)E[X_1]\left(1 + B\left(\frac{1}{2}\right)E[X_2]\left(1 + B\left(\frac{1}{3}\right)E[X_3]\,(1 + ...)\right)\right)\right] \tag{30}$$

$$= E\left[1 + B\left(\frac{1}{1}\right)X_1\left(1 + B\left(\frac{1}{2}\right)X_2\left(1 + B\left(\frac{1}{3}\right)X_3\,(1 + ...)\right)\right)\right] \tag{31}$$

$$\tag{32}$$

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

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

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. It is like assuming 2 induction hypotheses at the same time and proving both inductions steps at the same time vs doing separate induction proofs. Which should be easier because you have accesses to more assumptions at the same time.

Example 2.4.2 (coupled recursion)
Lets say you are interested in calculating the sensitivity of the solution of an ODE to a parameter a:

$$y' = ay, y(0) = 1 \Rightarrow \tag{33}$$
$$\partial_a y' = y + a\partial_a y \tag{34}$$

Turn (33) and (34) into RRVEs. To emphasize that they are coupled, that they should recurse together we write them in a matrix equation:

$$\begin{bmatrix} Y(t) \\ \partial_a Y(t) \end{bmatrix} = X(t) = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} a & 0 \\ 1 & a \end{bmatrix} X(Ut). \tag{35}$$

Notice how this gets rid of the additive branching recursion of equation (34).

Python Code 2.4.3 (implementation of (35))

```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 2.4.4 (coupled recursion)
Example (2.4.2) is inspired by [VSJ21].

Technique 2.4.5 (recursion in recursion)
Recursion in recursion is what is sounds like. It is like proving an induction step of an induction proof with induction.

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 support recursion but this comes with restrictions like maximum recursion depth and performance issues. When possible tail recursion is a way to implement recursion that solves those issues.

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 (35))
We implement (35) but this time with tail recursion. We collect addition operations in a vector *sol* and multiplication in a matrix W.

```
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

Technique 2.5.1 (SALT)
SALT

Example 2.5.2 (SALT wavelet)

check period 3

The current approach to non-linearity is inefficient

Example 2.5.3 (fixpoint problem)

check period 1

Branching tail recursion is little studied and hard. There are multiple ways to do branching tail recursion with each their advantages and disadvantages. In the context of recursive MC there are 2 techniques that stand out.

Technique 2.5.4 (tree regrowing)

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. Tree regrowing 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.

Technique 2.5.5 (backward tail recursion)

One way of doing branching tail recursion is by using operation buffers for all leafs which is not memory friendly. In backward tail recursion you retrace steps and do all operations in reverse to recover the buffer needed.

Related Work 2.5.6 (branching tail recursion)

Tree regrowing and backward tail recursion ideas appear in [VSJ21]. This blog discusses branching tail recursion: https://jeroenvanwijgerden.me/post/recursion-1/.

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. Before defining green functions we do some examples.

Example 3.1.1 ($y' = y$ average condition)

Let's solve

$$y' = y. \tag{36}$$

but this time with the following condition:

$$\int_0^1 y(s)ds = e - 1. \tag{37}$$

This still has solution $y(t) = e^t$. We define the corresponding source green function $G(t, x)$ for y' and this type of condition as follows:

$$G' = \delta(x - t), \int_0^1 G(s, x)ds = 0. \tag{38}$$

Solving this obtains:

$$G(t, x) = H(t - x) + x - 1. \tag{39}$$

Note that we could have used a different green function corresponding to a different linear differential operator.

It shouldn't be clear from this point but with this green function we form following integral equation for (36):

$$y(t) = e - 1 + \int_0^1 G(t,s)y(s)ds. \tag{40}$$

Turning equation (40) into a RRVE with recursive MC gives:

$$Y(t) = e - 1 + 2B\left(\frac{1}{2}\right)Y(S)(H(t-S) + S - 1). \tag{41}$$

With $S \sim U$. We will be skipping over the python implementation of equation (41) because it adds nothing new. Instead we plot realizations of equation (41) in Figure 4.

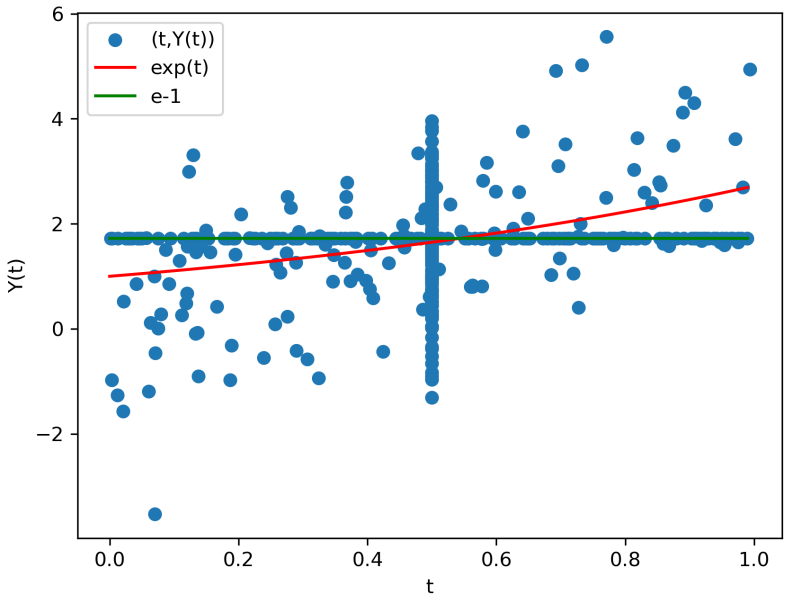

Figure 4: Recursive calls of (41) when calling $Y(0.5)$ 300 times. Points accumulate on the green line due to the Russian roulette, and at $t = 0.5$ because it is the starting value of the simulation.

Example 3.1.2 ($y'' = y$ mixed boundary conditions)
Lets solve the following boundary problem:

$$y'' = y, y(0) = 1, y'(1) = e. \tag{42}$$

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

$$G'' = \delta(t-x), G(0) = 0, G'(1) = 0. \tag{43}$$

Solving this obtains:

$$G(t,x) = \begin{cases} -t & \text{if } t < x \\ -x & \text{if } t \geq x \end{cases}. \tag{44}$$

The boundary green function $P(t,x)$ (for $x \in \{0,1\}$) for y'' and Dirichlet/Neumann boundary conditions is defined the following way:

$$P'' = 0, (P(0,x), P'(1,x)) = \begin{cases} (1,0) & \text{if } x = 0 \\ (0,1) & \text{if } x = 1 \end{cases}. \tag{45}$$

Which is just a basis for the homogenous solutions for now. Solving this gives:

$$P(t,x) = \begin{cases} 1 & \text{if } x = 0 \\ t & \text{if } x = 1 \end{cases}. \tag{46}$$

Again it shouldn't be clear from this point but with these set of green functions we form following integral equation for (42):

$$y(t) = P(t,0)y(0) + P(t,1)y(1) + \int_0^1 G(t,s)y(s)ds. \tag{47}$$

Equation (47) looks like:

$$Y(t) = 1 + te + lB\left(\frac{1}{l}\right) G(t,S)Y(S). \tag{48}$$

With $S \sim U$ and $l > 1 \in \mathbb{R}$. We visualize equation (48) on Figure 5.

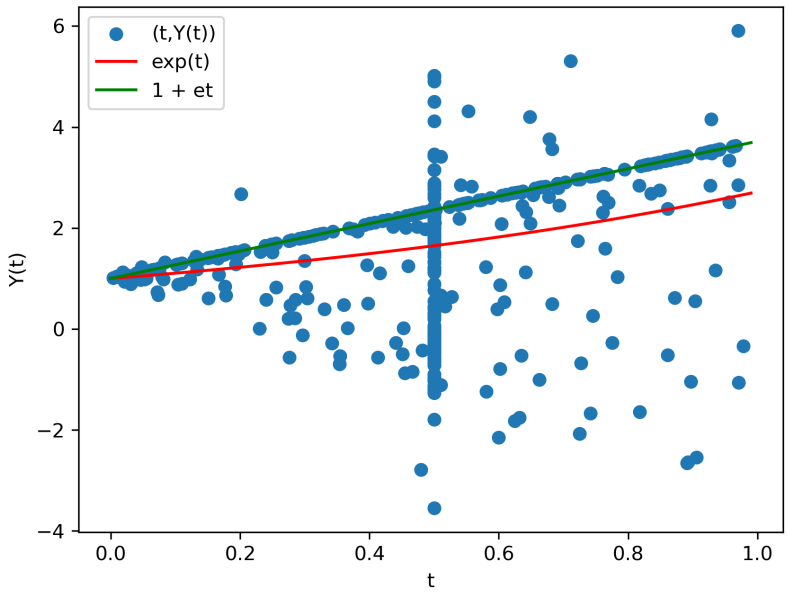

Figure 5: Recursive calls of (48), $l = 2$ when calling $Y(0.5)$ 300 times.

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 that we use to solve linear problems with linear conditions. The Green's function is the kernel that we need to put in front of the linear conditions or the source term that we integrate over and to obtain the solution and the green function has the property that it satisfies 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 acquired in the last subsection are Fredholm integral equations of the second kind. In this subsection we introduce coupled splitting a technique for RMC for this kind of equations.

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

$$\varphi(t) = f(t) + \lambda \int_a^b K(t,s)\varphi(s)ds. \tag{49}$$

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

If both K and f are nice, then for sufficiently small λ, it is straightforward to establish the existence and uniqueness of solutions for Fredholm equations of the second kind using a fix point argument.

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

$$y'' = y, y(b_0), y(b_1). \tag{50}$$

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

$$P(t,x) = \begin{cases} \frac{b_1 - t}{b_1 - b_0} & \text{if } x = b_0 \\ \frac{t - b_0}{b_1 - b_0} & \text{if } x = b_1 \end{cases} \tag{51}$$

$$G(t,s) = \begin{cases} -\frac{(b_1 - t)(s - b_0)}{b_1 - b_0} & \text{if } s < t \\ -\frac{(b_1 - s)(t - b_0)}{b_1 - b_0} & \text{if } t < s \end{cases}. \tag{52}$$

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

$$y(t) = P(t,b_0)y(b_0) + P(t,b_1)y(b_1) + \int_{b_0}^{b_1} G(t,s)y(s)ds \tag{53}$$

$$Y(t) = P(t,b_0)y(b_0) + P(t,b_1)y(b_1) + lB\left(\frac{1}{l}\right)(b_1 - b_0)G(t,S)y(S). \tag{54}$$

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

Example 3.2.3 (coupled splitting on (3.2.2))
Next to normal splitting (2.1.6) we can also split the domain in equation (53):

$$y(t) = P(t, b_0)y(b_0) + P(t, b_1)y(b_1) + \frac{1}{2}\int_{b_0}^{b_1} G(t, s)y(s)ds + \frac{1}{2}\int_{b_0}^{b_1} G(t, s)y(s)ds \tag{55}$$

$$y(t) = P(t, b_0)y(b_0) + P(t, b_1)y(b_1) + \int_{b_0}^{\frac{b_1+b_0}{2}} G(t, s)y(s)ds + \int_{\frac{b_1+b_0}{2}}^{b_1} G(t, s)y(s)ds \tag{56}$$

Coupling can get rid of the additive branching recursion in the RRVEs corresponding to (55) and (56). Resulting in following RRVE:

$$X(t_1, t_2) = \begin{bmatrix} P(t_1, b_0) & P(t_1, b_1) \\ P(t_2, b_0) & P(t_2, b_1) \end{bmatrix} \begin{bmatrix} y(b_0) \\ y(b_1) \end{bmatrix} + W \begin{bmatrix} G(t_1, S_1) & G(t_1, S_2) \\ G(t_2, S_1) & G(t_2, S_2) \end{bmatrix} X(S_1, S_2). \tag{57}$$

With W the right weighting matrix (see code (3.2.4) for an example) and S_1, S_2 can be chosen in various ways.

Python Code 3.2.4 (implementation of (57))
We implemented equation (57) in example (3.2.3) with recursion but in this case it is actually possible to implement it forwardly because the time proces is nice.

```
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
```

Figure 6 resembles a fixed point iterations, leading us to hypothesize that coupled splitting can achieve convergence in most cases where a fixed point argument holds true and that 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 usually done when approaching the bottleneck. Alternatively when reaching the bottleneck it is possible to rely on MC convergence.

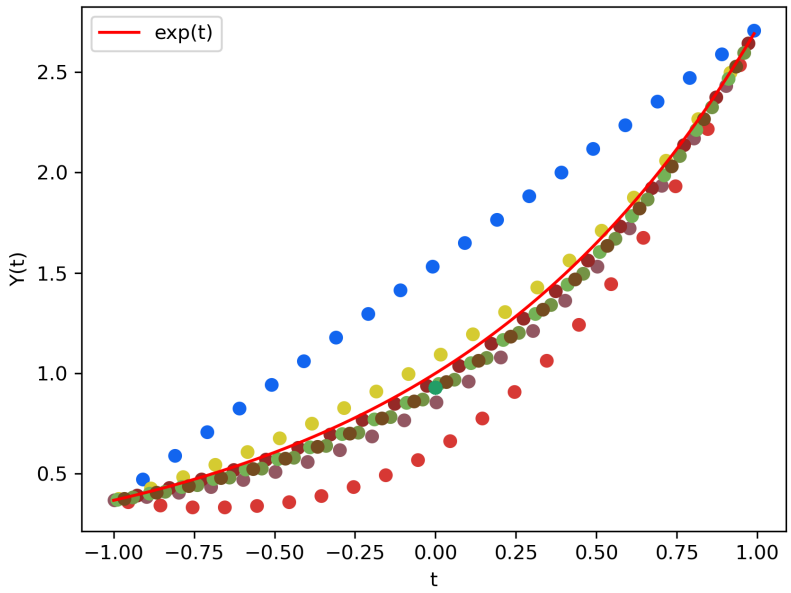

Figure 6: Recursive calls of equation (57) when calling $X(0)$ once, with a split size of 20, S_j coupled such they're equally spaced (they don't have to be independent) and coupling is colored. The initial conditions for this call are $y(-1) = e^{-1}$ and $y(1) = e^1$, with Russian roulette rate $l = 1.2$.

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.

Example (3.2.3) isn't the best example to demonstrate coupled splitting. We don't exploit locality and smoothness of the problem. We conjecture that this algorithm is useful for linear Fredholm equations of the second kind in cases where MC integration win over classic integration: high dimensional, non-smooth kernels or nasty domains.

Related Work 3.2.5 (coupled splitting)
Coupled splitting is partly inspired by how [SM09] reduces variance by using a bigger submatrices . See [GH21] for a discussion on convergence of recursive stochastic algorithms. We highly recommend watching the corresponding video [Abh20] before reading the paper.

3.3 Initial Value Problems

Right now we don't have a RMC algorithm for IVPs that can guarantee a reasonable variance or even existence when increasing the time domain. Classing IVP solvers rely on shrinking the time steps for convergence. Recursion in Recursion MC (RRMC) for IVPs tries to emulate this behavior.

Example 3.3.1 (RRMC $y' = y$)
Let's us explain RRMC for IVPs with our main example. Imagine we have a time

stepping sheme (t_n) $(t_n > t_{n-1})$ then following integral equations hold:

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

Turn these in following class of RRVEs:

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

A problem with these RRVEs is that we don't know $y(t_n)$. Instead we can replace it with an unbiased estimate y_n which we keep frozen:

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

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

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 to be $O\left(\frac{h^{1.5}}{\sqrt{\text{nsim}}}\right)$.

1.5 order of convergence is cool but this begs the question on how to achieve higher order of convergence in h. Again it is easy to imitate classical methods to achieve higher order convergence. We do this by removing lower order terms (which requires smoothness) with control variates like the MC trapezoidal rule (2.2.2).

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

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

We need a lower order approximation of the integrand:

$$y(s) = y(t_n) + (s - t_n)y'(t_n) + O((s - t_n)^2) \tag{63}$$

$$\approx y(t_n) + (s - t_n)f(y(t_n), t_n) \tag{64}$$

$$\approx y(t_n) + (s - t_n)\left(\frac{y(t_n) - y(t_{n-1})}{t_n - t_{n-1}}\right) \tag{65}$$

$$\approx y(t_n)(1 + s - t_n). \tag{66}$$

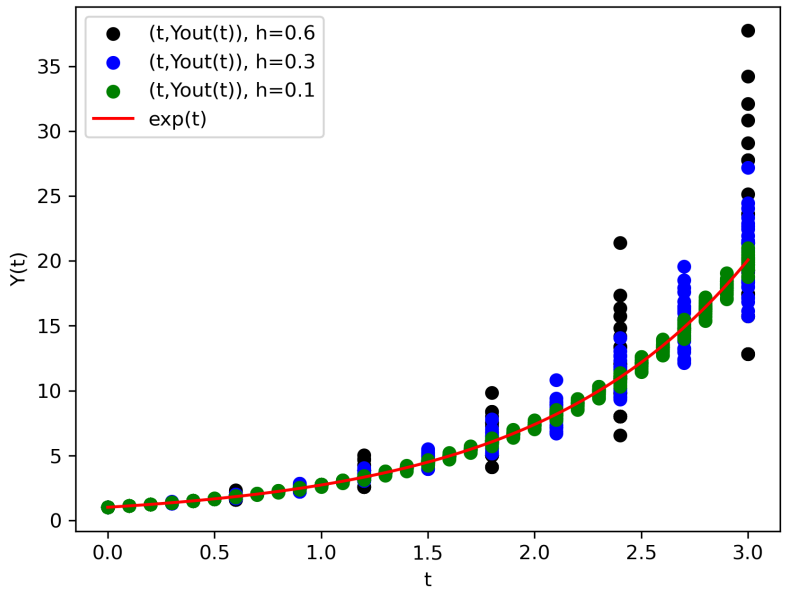

Figure 7: Recursive calls of equation (61) when calling $Y_{\text{out}}(3, h)$ 30 times for different h.

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 won't 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 info from the outer recursions, this doesn't mean that reducing root mean square error by splitting doesn't work, you just have to be careful with the bias. We conjecture that the bias in RRMC converges faster then the variance.

Example 3.3.5 (nonlinear RRMC IVP)

Consider:

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

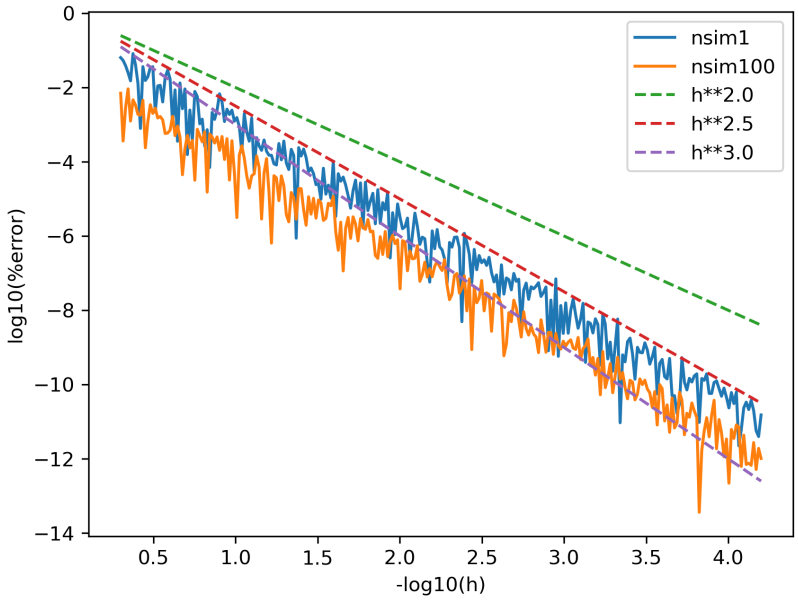

Figure 8: Log-log plot of example (3.3.3).

with solution: $y(t) = t^2$. With integral equation:

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

control variating $y^2(s)$ up to second order (via Taylor):

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

Then we have to integrate the control variate:

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

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
```

3.4 Limitations and Future Work

RMC convergence issues, exploding variance problems quasi RRMC

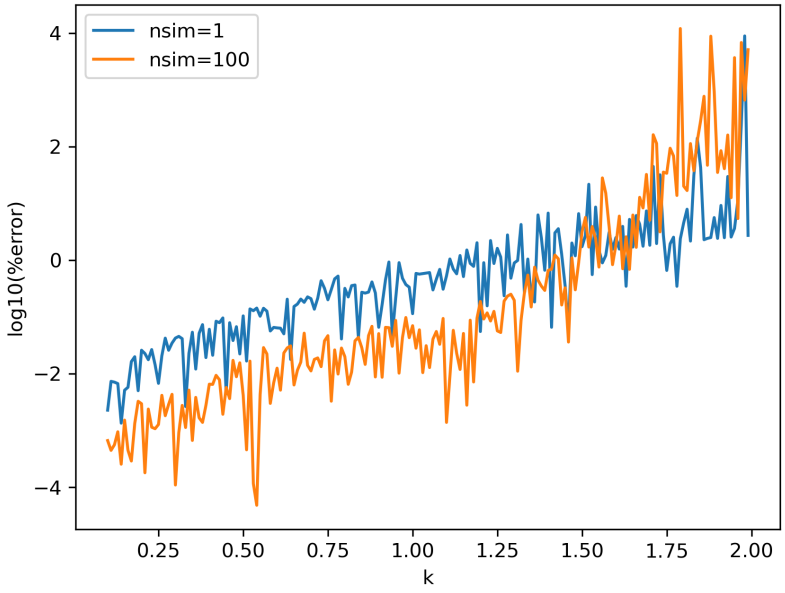

Figure 9: The logarithmic percentage error of $Y(0)$ for (54), with $l = 1.2$ and initial conditions $y(-k) = e^{-k}$ and $y(k) = e^{k}$, displays an exponential increase until approximately $k = 1.5$, beyond which additional simulations fail to reduce the error, indicating that the variance doesn't exist.

Similarly to classic methods, RRMC struggles with big negative coefficients in front of the recursive parts. DRRMC is a potential solution to this problem but we don't think it is effective.

Definition 3.4.1 (DRRMC)
Consider a general linear ODE IVP problem:

$$x' = Ax + g, x(0) = x_0. \tag{76}$$

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

$$x' + Dx = (A + D)x + g. \tag{77}$$

Following integral equation can be derived by using integrating factor:

$$x(t) = e^{D(t_n - t)}x(t_n) + \int_{t_n}^{t} e^{D(s-t)}(A + D)x(s)ds + \int_{t_n}^{t} e^{D(s-t)}g(s)ds. \tag{78}$$

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

$$\int_{t_n}^{t} e^{D(s-t)}(A + D)x(t_n)ds = D^{-1}(I - e^{D(t_n - t)})(A + D)x(t_n). \tag{79}$$

Note that D may be chosen differently every recursion.

Example 3.4.2 (DRRMC)
Consider

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

With

$$A = \begin{pmatrix} 0 & 1 \\ -1000 & -1001 \end{pmatrix}.$$

we choose D fixed over all outer recursions:

$$D = \begin{pmatrix} 1 & 0 \\ 0 & 1000 \end{pmatrix}.$$

Related Work 3.4.3 (DRRMC)
DRRMC is inspired by $\bar{\sigma}$ parameter in [Saw+22] but instead of importance sampling and weight window 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. Similar manipulations can also be found in exponential integrator methods.

Example 3.4.4 (random ODEs)
Let's solve:

$$Y' = AY, Y(0) = 1, A = \text{Uniform}(0, 1). \tag{80}$$

With solution $Y(t) = e^{tA}$ which is a RV. With the techniques we have right now we can simulate "unbiased estimates of simulations of the solution" X. With these you can get $E[f(Y(t))]$ (with f analytic) to the problem:

$$\begin{aligned} E[f(Y(t))] &= E[E_A[f(Y(t)) \mid A]] && (81)\\ &= E[E_A[f(Y(t))]] && (82)\\ &= E[E_A[f(E_x[X(t, A)])]] && (83)\\ & && (84) \end{aligned}$$

We can estimate $f(E_x[X(t, A)])$ like (2.3.3).

In our example we estimate the expectance and variance which can be derived from the solution:

$$E_A[Y(t)] = \frac{e^t}{t} - \frac{1}{t} \tag{85}$$

$$E_A[Y^2(t)] = \frac{e^{2t}}{2t} - \frac{1}{2t} \tag{86}$$

Python Code 3.4.5 (implementation of (3.4.4))

```
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},", f"%error = {(sol -
    s)/s}")
print(f"E(Y^2({t})) is approx {sol2},", f"%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
```

IVPs solver problems slow and instable, maybe special cases. CV in the inner recursion example with recursing on DY
notion of green function is vague maybe not the right tools mention mixed boundary work

4 Brownian Motion

Current RMC algorithms for PDEs are linked to Brownian motion. In this section build up to recursive first passage sampling which is similar to walk on sphere.

4.1 Heat Equation

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

Definition 4.1.1 (Brownian motion)
Define Brownian motion W_t as the limit/logical generalization when $n \to \infty$ of following discrete proces defined as:

$$\begin{cases} X_t^n = X_{t-\frac{1}{n}}^n + Z_n \\ X_0^n = 0 \end{cases} . \tag{87}$$

With $Z_n \sim N(0, \frac{1}{n})$ i.i.d . From this definition it is easily seen that $W_t \sim N(0, t)$.

Lemma 4.1.2 (self-affinity Brownian motion)
Brownian motion is self affine as a random proces that means you can cut a small part of it move so it starts in 0 and scale time to the original size and space such that the variance stays the same to get back the whole Brownian motion.

$$\forall c \in \mathbb{R}_0^+ : \frac{W_{ct}}{\sqrt{c}} \sim W_t. \tag{88}$$

Definition 4.1.3 (1D heat equation Dirichlet)
We define the 1D heat equation for u on connected domain Ω with Dirichlet boundary conditions the following way:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2}. \tag{89}$$

Given $u(x,t) = \psi(x,t), \forall (x,t) \in \partial\Omega : t < \sup\{t|(x,t) \in \Omega\}$.
An example how the Dirichlet condition is defined for a domain is the parabola on Figure 10 but reverse in time.

Lemma 4.1.4 (Brownian motion and the heat equation)
For problem (4.1.3) if $|\psi| < \infty$ there is following formula:

$$u(x,t) = E[\psi(Y_\tau, \tau)|Y_t = x]. \tag{90}$$

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:

$$\frac{u(x,t) - u(x,t-\Delta t)}{\Delta t} = \frac{u(x+\Delta x,t) - 2u(x,t) + u(x-\Delta x,t)}{\Delta x^2}. \tag{91}$$

Isolate $u(x,t)$:

$$u(x,t) = \frac{\Delta t}{2\Delta t + \Delta x^2}\left(u(x+\Delta x,t) + u(x-\Delta x,t)\right) + \frac{\Delta x^2}{2\Delta t + \Delta x^2}\left(u(x,t-\Delta t)\right). \tag{92}$$

Because $u(x+\Delta x,t) \approx u(x-\Delta x,t) \approx u(x,t-\Delta t) \approx$ RHS of equation (92) we may Russian roulette to remove branching recursion and generate a recursion path instead of a tree.

$$Z(x,t) = \begin{cases} \psi(\text{argmin}_{b\in\partial\Omega}||(x,t)-b||) & \text{when } (x,t) \notin \Omega \\ \begin{cases} Z(x+\Delta x,t) & \text{with chance } \frac{\Delta t}{2\Delta t+\Delta x^2} \\ Z(x-\Delta x,t) & \text{with chance } \frac{\Delta t}{2\Delta t+\Delta x^2} \\ Z(x,t-\Delta t) & \text{with chance } \frac{\Delta x^2}{2\Delta t+\Delta x^2} \end{cases} & \text{else .} \end{cases} \tag{93}$$

This is a RRVE, Z has finite variance because $|\psi| < \infty$ and $E[Z(x,t)]$ is the solution to the discretized heat equation. Taking the limit makes the discrete solution go

the the real solution. For (93) the limit makes the recursion path go to Brownian motion Y_t.

$$Z(x,t) \to \psi(Y_\tau, \tau). \tag{94}$$

Finishing the proof. □

Related Work 4.1.5

(4.1.4) is a subcase of the Feynman-Kac formula. For a proof and a 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 (4.1.4) 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

$$\text{FPt}(X_t, S) = \inf\{t | (X_t, t) \notin V\}. \tag{95}$$

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

$$\text{FP}(X_t, S) = (X_\tau, \tau), \tau = \text{FPt}(X_t, V). \tag{96}$$

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 (4.1.4). □

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 on Figure 10.

Lemma 4.2.5

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

$$V_1 \subset V_2 \Rightarrow \text{FPt}(X_t(\omega), V_1) \leq \text{FPt}(X_t(\omega), V_2). \tag{97}$$

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 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.

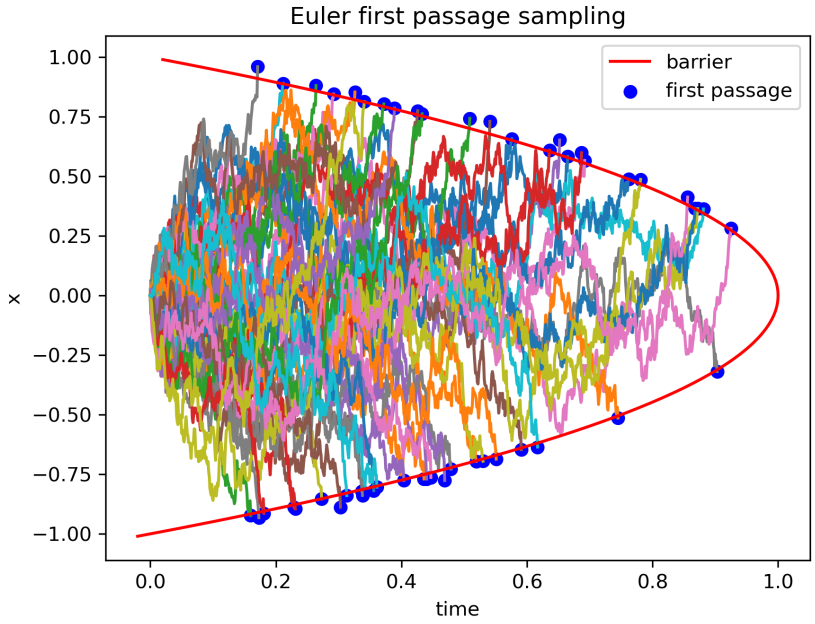

Figure 10: 50 realizations of Euler first passage sampling with step size 0.001.

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 (4.1.2) and using the fact that a parabola domain is convex. To dampen barrier overstepping of (4.2.8) we use a smaller scaling then 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
```

```python
# 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)
```

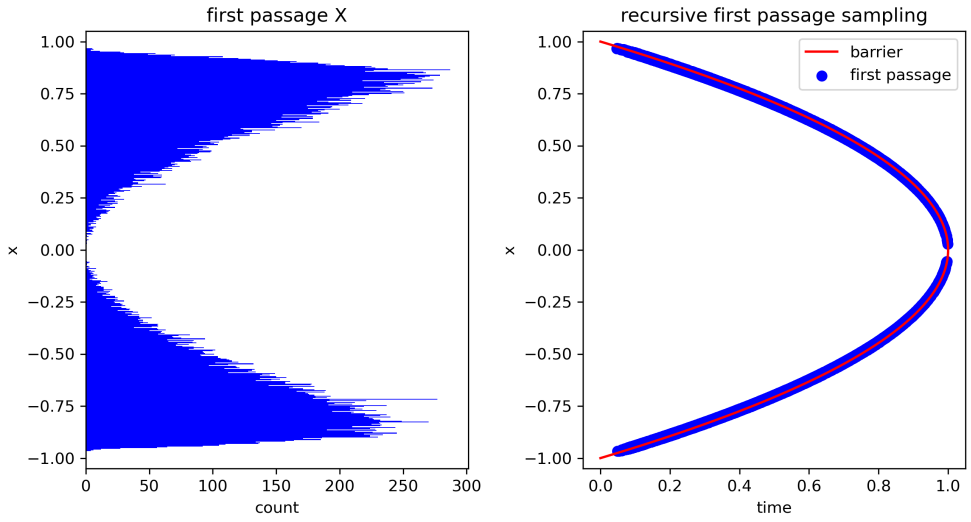

Figure 11: 50000 of realizations of recursive first passage sampling produced by (4.2.9). The precomputed sample of 5000 first passages of a triangular barrier uses the Euler scheme with step size 0.001.

Related Work 4.2.10 (recursive first passage sampling)
The original walk on spheres is a recursive first passage algorithm. Recursive first passage sampling gets discussed in [HT16]. 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 scales and translates with our base first passage sampler.

Technique 4.2.12 (Brownian motion path stitching)
Instead of thinking in sampling 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 "stich" 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 then 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

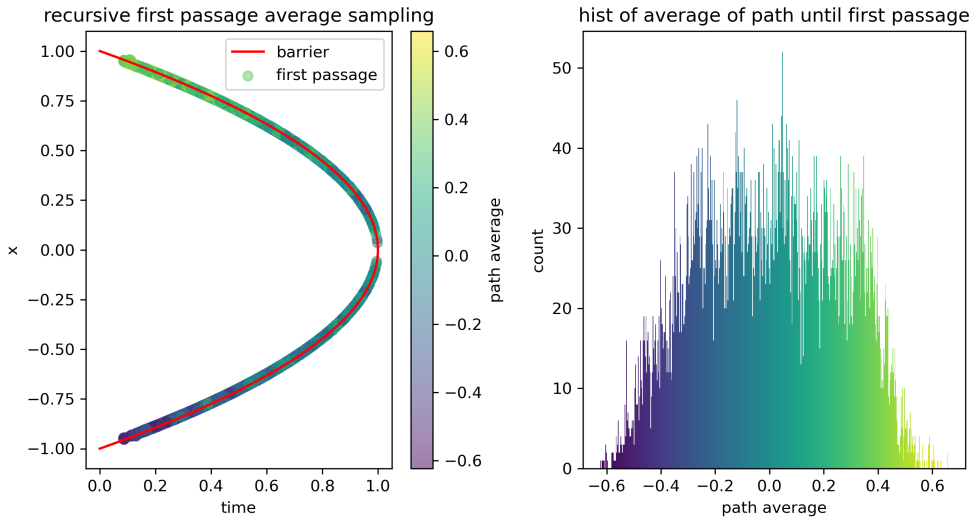

Figure 12: 10000 of realizations of recursive first passage sampling (for the average). The precomputed sample of 1000 first passages and averages of a triangular barrier uses the Euler scheme with step size 0.001.

inhomogeneous and 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.

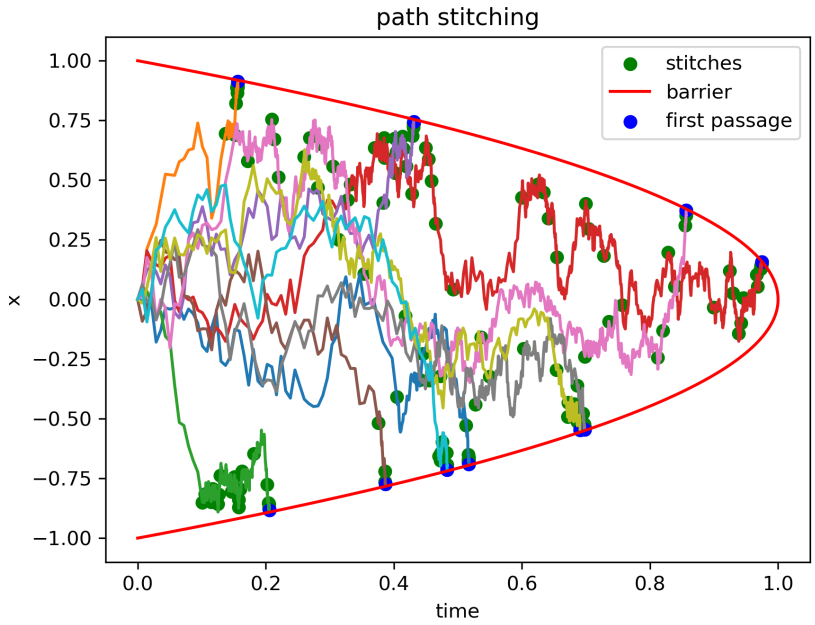

Figure 13: 10 paths build with path stitching build out of precomputed Euler scheme generated paths with step size 0.01.

Related Work 4.2.14 (path stitching)
Path stitching appears frequently in rendering and also in [Das+15] and [JL12].

4.3 Limitations and Future Work

source term to (4.1.4) different from Feynman kac
extending first passage averages sampling to geometric brownian motion + discrete scaling idea
recursive passage sampling in higher dimension

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).

[Gil13] Michael B. Giles. *Multilevel Monte Carlo methods*. en. arXiv:1304.5472 [math]. Apr. 2013. URL: http://arxiv.org/abs/1304.5472 (visited on 12/04/2022).

[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).

[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).

[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).

[Vea] Eric Veach. “ROBUST MONTE CARLO METHODS FOR LIGHT TRANSPORT SIMULATION”. en. In: ().

[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).