Masterproef scriptie

Unbiased Monte Carlo for Recursive Integrals

Auteur: *Isidoor Pinillo Esquivel*

Promotor: *Wim Vanroose*

Academiejaar 2022-2023

Contents

Abstract

We will write this at the end.

1 Introduction

1.1 Introductory Example

To get familiar with Monte Carlo for estimating recursive integrals we demonstrate it on following problem:

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

By integrating both sides of (1) following integral equation can be derived:

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

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

$$Y(t) = 1 + ty(Ut).$$

where $U = \text{Uniform}(0, 1)$. 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:

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

Equation (3) is a recursive random variable equation (RRVE). If you would implement equation (3) 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 (3))

```
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
```

An issue with (1.1.1) is that the variance increases rapidly when t increases. Which we later solve in the section on ODEs. Note that (1.1.1) keeps desirable properties from unbiased Monte Carlo methods such as: being embarrassingly parallel, robustness and having simple error estimates.

1.2 Contributions

We write this at the end. Probably a lot of conjectures.

1.3 Related Work

work on

- alternative methods for recursive integrals
- MC work on ODEs
- MC work on PDEs
- WoS

This is just to give a general overview we probably reference specific ideas when we first introduce them.

2 Background

2.1 Modifying Monte Carlo

Once we have a RRVE it is possible to modify it to have more desirable properties.

Russian roulette is a Monte Carlo technique widely used in rendering. The idea behind Russian roulette is replacing a random variable with a cheaper 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:

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

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

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

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

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

where $B(\frac{1}{100}) \sim \text{Bernouli}(\frac{1}{100})$. Now $\tilde{Z}$ just requires calling f on average once every 100 simulations with the variance only increasing slightly compared to Z.

Maybe this wasn't the best example because you could also estimate the expectance of the 2 terms of Z separately.

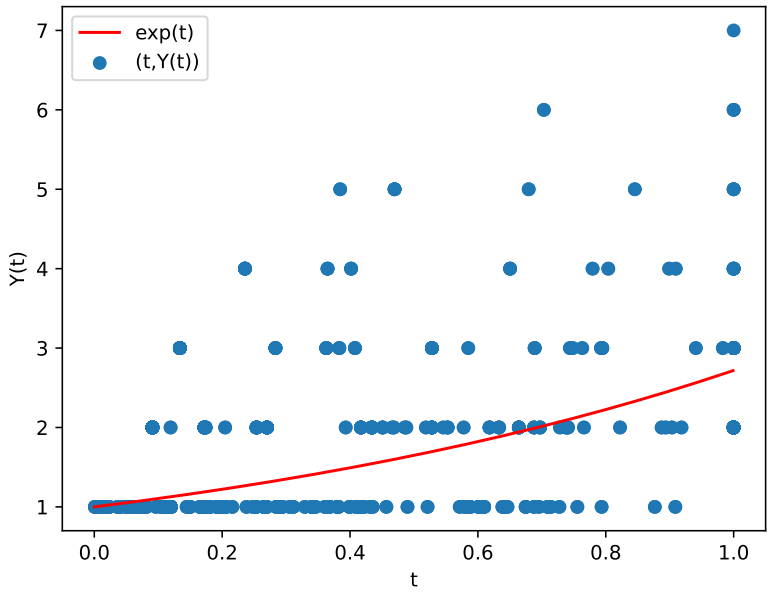
Example 2.1.3 (Russian roulette on (3))
Russian roulette can fix the indefinite recursion issue of equation (3) by approximating Y near $t = 0$ with 1. Concretely we replace the t in front of the recursive term with $B(t) \sim$ Bernouli(t) when $t < 1$.

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

Python Code 2.1.4 (Russian roulette on (3))

```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
```

An intrestring property of this algorithm is that $Y(t) \in \mathbb{N}$. A way to visualize this is by plotting every recursion call $(t, Y(t))$.



Splitting is a technique that has almost the reverse effect as Russian roulette. The idea behind splitting is to reduce variance in certain places by using more samples.

Definition 2.1.5 (splitting)
Splitting X means using multiple $X_j \sim X$ not independent per se to lower variance

by averaging them:

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

Splitting the recursive term in a RRVE can lead to (additive) branching recursion. Extra care should be taken that all branches get terminated with probability 1. This can be achieved by termination strategies already discussed and later we discuss coupled recursion for alleviating additive branching recursion in RRVEs.

Example 2.1.6 (splitting on (3))
We can "split" the recursive term of (3) in 2:

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

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

Python Code 2.1.7 (splitting on (3))

```
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 2.1.8 (2-level MC)
2-level MC on X with parameters $\tilde{X}, Y : E[\tilde{X}] = E[Y]$:

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

Definition 2.1.9 (control variates)
Control variate on $f(X)$ is

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

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

Example 2.1.10 (control variate on (3))
To make a control variate for (3) that reduces variance we use following approximation of $y(t) \approx 1 + t$:

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

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

Python Code 2.1.11 (control variate on (3))

```python
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

Our favorite work that discusses these techniques is [Vea]. More interesting works can be found on Monte Carlo techniques in rendering. 2-level gets discussed in [Gil13].

2.2 Monte Carlo Trapezoidal Rule

We present here a Monte Carlo trapezoidal rule with similar convergence behavior to methods discussed later. The Monte Carlo trapezoidal rule will just be regular Monte Carlo 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:

$$\int_x^{x+dx} f(s)ds \approx \frac{f(x)+f(x+dx)}{2} + f(S_x) - f(x) - \frac{S_x - x}{dx}(f(x+dx) - f(x)) \quad (4)$$

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:

$$\int_a^b f(s)ds \approx \quad (5)$$

$$\sum_x \frac{f(x)+f(x+dx)}{2} + lB\left(\frac{1}{l}\right)\left(f(S_x) - f(x) - \frac{S_x - x}{dx}(f(x+dx) - f(x))\right) \quad (6)$$

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
```

```python
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
```

What is surprising about this MC composite rule is that under the right smoothness conditions it adds 0.5 for every dimension order of convergence over the normal composite rule.

Lemma 2.2.4 (half variance phenomenon)
Maybe a lemma about MC integrating a polynomial with proof and this becomes a theorem

Proof. Also a maybe, maybe just a numerical example. □

Related Work 2.2.5
Optimal theoritcal bounds on randomized algorithms can be found in: (see literature randomized trapezoidal rule) [Wu20].

comparing normal vs Monte Carlo trapezoidal rule and highlighting the "half variance phenomenon". + maybe integrating polynomials for intuition

2.3 Unbiased Non-Linearity

At first sight it 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 practically 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:

$$y' = y^2. \tag{7}$$

with $y(1) = -1$. This has solution $-\frac{1}{t}$. Integrate both sides of equation (7) to arrive at following integral equation:

$$y(t) = -1 + \int_1^t y(s)y(s)ds. \tag{8}$$

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

$$Y(t) = -1 + (t-1)Y_1(S)Y_2(S).$$

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:

$$e^{E[X]} = \sum_{n=0}^{\infty} \frac{E^n[X]}{n!} \tag{9}$$

$$= 1 + \frac{1}{1}E[X]\left(1 + \frac{1}{2}E[X]\left(1 + \frac{1}{3}E[X]\,(1 + ...)\right)\right). \tag{10}$$

Change the fractions of equation (10) to Bernoulli processes and replace all X with independent X_j with $E[X] = E[X_i]$.

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

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 $e^{\int_0^t s^2 ds}$:

```
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
```

```python
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

A similar approach to non-linearity can be found in [ET19]. We have more papers on how to deal with non-linearity stashed, no idea if they are worth mentioning.

2.4 Recursion

In this section we discuss recursion related techniques.

Technique 2.4.1 (coupled recursion)

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

$$y' = ay, y(0) = 1 \Rightarrow \tag{11}$$
$$\partial_a y' = y + a\partial_a y' \tag{12}$$

Turn (11) and (12) into RRVEs. To emphasize that they are coupled, that they should recurse together we write them in a matrix equation:

$$\begin{bmatrix} Y(t) \\ \partial_a Y(t) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} a & 0 \\ 1 & a \end{bmatrix} \begin{bmatrix} Y(Ut) \\ \partial_a Y(Ut) \end{bmatrix}. \tag{13}$$

Python Code 2.4.3 (implementation of (13))

```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]
```

Technique 2.4.4 (recursion in recursion)

Recursion in recursion is what is sound like. This is like proving a induction step of an induction proof with induction.

Example 2.4.5 (recursion in recursion)
maybe induction in induction proof example or a reference to ODE solvers later.

Related Work 2.4.6 (recursion in recursion)
The next flight variant of WoS is a beautiful example of recursion in recursion described in [Saw+22].

Most programming languages support recursion but this comes with restrictions like maximum recursion depth and performance issues. Tail recursion solve those issues when possible.

Technique 2.4.7 (non-branching tail recursion)
Tail recursion is reordering all operations in a way that almost no operation needs to happen after the recursion call such that when reaching the last recursion call we can return the answer without retracing all steps.

All non-branching recursion in this paper can be implemented straight forwardly. This can easily be seen because all of the operations are associative ($(xy)z = x(yz)$). Tail recursion is not always desirable because we lose the intermediate values of the recursion calls. It is also possible to combine tail recursion with normal recursion.

Python Code 2.4.8 (tail recursion on (13))
We implement (13) 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]
```

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

Technique 2.4.9 (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 2.4.10 (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 2.4.11 (branching recursion)
This blog discusses branching tail recursion: `https://jeroenvanwijgerden.me/post/recursion-1/`. The techniques for tail recursion gets discussed in [VSJ21]. They applied it on a non-branching estimator, which we think is overkill.

2.5 Green Functions

green function stuff that we will be needing, we aren't sure in how much detail we're going to go.

Example 2.5.1 (numerical green functions)
There will be probably some green functions that we need that don't have an analytic expression yet.

3 1-Dimensional Recursive Integrals

3.1 Linear Recursive Integrals

We have algo in mind for this case based on coupled recursion on disjunct sets.

3.2 IVPs ODEs

An IVP example probably using DRRMC maybe compare it to parareal. Maybe also non-linear algo

3.3 BVPs ODEs

A BVP example using yet another algo that hopefully has the half variance phenomenon.

4 Higher Dimensional Recursive Integrals

4.1 Complicated Geometry

Example 4.1.1 (nasty 2D integral)
2D integral that is difficult because of its geometry

4.2 Recursive Brownian Motion

WoS like way to simulate Brownian motion which is related to the green function of the heat equation

Example 4.2.1 (recursive Brownian motion)
see period5

4.3 Heat Equation

a geometric robust way to solve the heat equation and maybe a higher order method to solve the heat equation

4.4 Wave Equation

probably won't get to it

References

[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).

[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).

[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).

[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).

5 Appendix

Derivation of the green functions and some expressions.