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 section we introduce recursive Monte Carlo with following problem:

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

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:

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

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:

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

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 (2.2.1) 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
```

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

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

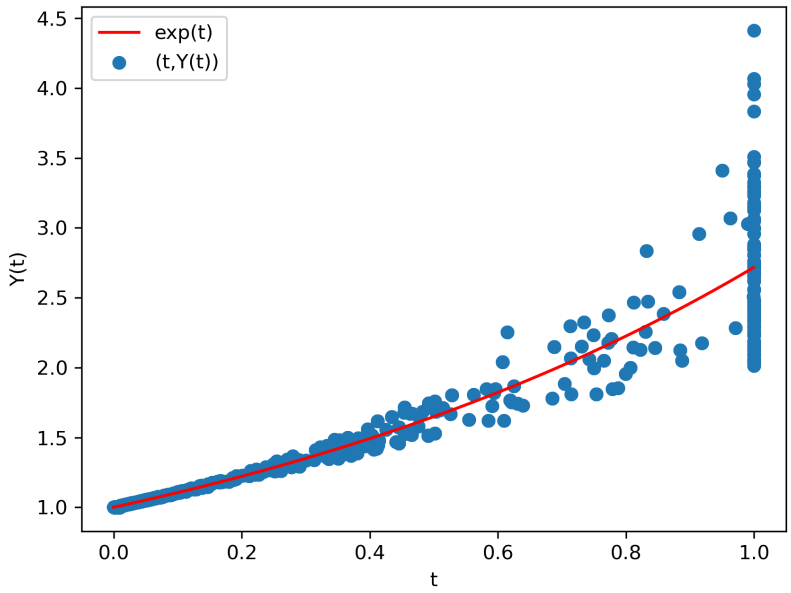

Figure 1: Recursive calls of (1.1.1)

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 Notation

Notations used in this thesis include:

- $U \sim \text{Uniform}(0, 1)$.
- $B(p) \sim \text{Bernoulli}(p)$.

- RV = random variable.
- RVs will be denoted with capital letters, e.g., X, Y or Z.
- RRVE (recursive RV equation): An equation that defines a family of random variables in terms of its self.
- MC = Monte Carlo.

2.2 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 2.2.1 (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 2.2.2 (Russian roulette)
Say that we are interested in estimating $E[Z]$ with Z defined in the following way:

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

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

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

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

In this example it is also possible to estimate the expectance of the 2 terms of Z separately.

Example 2.2.3 (Russian roulette on (3))
Russian roulette can fix the indefinite recursion issue of equation (3) by approximating Y near $t = 0$ with 1 sometimes. Concretely we replace the t in front of the recursive term with $B(t)$ when $t < 1$.

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

Python Code 2.2.4 (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
```

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

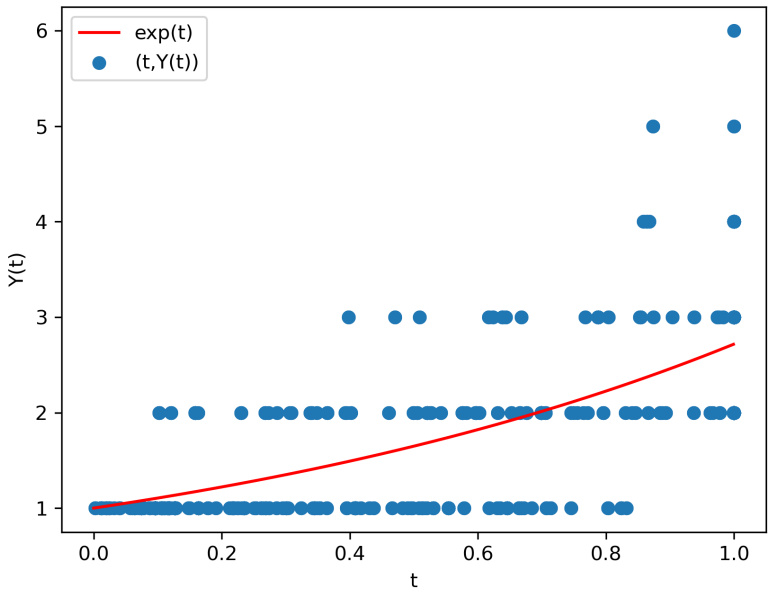

Figure 2: Recursive calls $(t, Y(t))$ of (2.2.4)

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 2.2.5 (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, 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.

Example 2.2.6 (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 2.2.7 (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 2.2.8 (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 2.2.9 (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 2.2.10 (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 2.2.11 (control variate on (3))

```
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 2.2.12
Our favorite work that discusses these techniques is [Vea]. More interesting works can be found on MC techniques in rendering. 2-level gets discussed in [Gil13].

2.3 MC Trapezoidal Rule

We present here 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 2.3.1 (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 2.3.2 (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 2.3.3 (implementation of (2.3.2))
We implement (2.3.2) 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(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
```

We conjecture that this MC composite rule adds 0.5 order of convergence over the normal composite rule under the right smoothness conditions.

Lemma 2.3.4 (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 2.3.5
Optimal theoretical bounds on randomized algorithms can be found in: (see literature randomized trapezoidal rule) [Wu20].

2.4 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 2.4.1 ($y' = y^2$)
Let's do following example:

$$y' = y^2, y(1) = -1. \tag{7}$$

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 $Y_1, Y_2 \sim Y$:

$$Y(t) = -1 + (t-1)Y_1(S)Y_2(S). \tag{9}$$

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

Python Code 2.4.2 ($y' = y^2$)

```
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 2.4.3 ($e^{E[X]}$)
$e^{\int x(s)ds}$ 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:

$$e^{E[X]} = \sum_{n=0}^{\infty} \frac{E^n[X]}{n!} \tag{10}$$

$$= 1 + \frac{1}{1}E[X]\left(1 + \frac{1}{2}E[X]\left(1 + \frac{1}{3}E[X]\left(1 + ...\right)\right)\right). \tag{11}$$

Change the fractions of equation (11) to Bernoulli processes and replace all X with independent X_j with $E[X] = E[X_i]$.

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

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

Python Code 2.4.4 ($e^{E[X]}$)
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
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 2.4.5
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.

2.5 Recursion

In this subsection we discuss recursion related techniques.

Technique 2.5.1 (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 2.5.2 (coupled recursion)
Lets say you are interested in calculating the sensitivity of the solution of an ODE to a parameter a:

$$y' = ay, y(0) = 1 \Rightarrow \tag{12}$$
$$\partial_a y' = y + a\partial_a y \tag{13}$$

Turn (12) and (13) into RRVEs. To emphasize that they are coupled, that they should recurse together we write them in a matrix equation:

$$\begin{bmatrix} Y(t) \\ \partial_a Y(t) \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} + t \begin{bmatrix} a & 0 \\ 1 & a \end{bmatrix} \begin{bmatrix} Y(Ut) \\ \partial_a Y(Ut) \end{bmatrix}. \tag{14}$$

Notice how this gets rid of the additive branching recursion of equation (13).

Python Code 2.5.3 (implementation of (14))

```
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 2.5.4 (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 2.5.5 (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. When possible tail recursion is a way to implement recursion that solves those issues.

Technique 2.5.6 (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 straightforwardly 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 2.5.7 (tail recursion on (14))
We implement (14) but this time with tail recursion. We collect addition operations in a vector *sol* and multiplication in a matrix W.

```python
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]
```

Related Work 2.5.8 (branching recursion)
This blog discusses branching tail recursion: `https://jeroenvanwijgerden.me/post/recursion-1/`. Interesting techniques like tail recursion gets discussed in [VSJ21].

3 ODEs

3.1 From ODEs to Integral Equations

In this subsection we discuss informally how to turn ODEs into integral equations.

Let's start from following form:

$$L(y) = f$$

where L is a linear operator, f be generic for now and some initial/boundary condition for y. All the methods we have in mind are some kind of integral transform:

$$y(t) = \int_{\Omega} \varphi(x) K(t,x) dx$$

where the integral may also be a summation and we may chose $K(t,x), \Omega$ and let $\varphi(x)$ be our new unknown. Note that we don't know that such representation exist or even is well defined but we continue and fix this in the future (hopefully). If you sub this into the first equation you obtain the following:

$$L\left(\int_{\Omega} \varphi(x) K(t,x) dx\right) = f \Leftrightarrow$$
$$\int_{\Omega} \varphi(x) L\left(K(t,x)\right) dx = f$$

this is a Fredholm integral equation of the first type in $\varphi(x)$ https://en.wikipedia.org/wiki/Fredholm_integral_equation if we had let Ω depend on t in a certain way we would have obtained a Voltera integral equation of the second kind https://en.wikipedia.org/wiki/Volterra_integral_equation.
Let's derive the method of source green functions in this framework https://en.wikipedia.org/wiki/Green%27s_function. This can be done by choosing the free things in a way something nice happens:

$$L\left(K(t,x)\right) = \delta(t-x)$$

with the same domain and initial/boundary conditions such that the ones of y hold. Choosing the initial/boundary conditions that way is easy but not trivial.
For a dirichlet boundary condition in s the following can be done:

$$y(s) = \int_\Omega \varphi(x)K(s,x)dx \Leftarrow$$
$$K(s,x) = \frac{y(s)l(x)}{\varphi(x)} \text{ and } 1 = \int_\Omega l(x)dx$$

For a Neumann boundary condition in s the following can be done:

$$y'(s) = \int_\Omega \varphi(x)K'(s,x)dx \Leftarrow$$
$$K'(s,x) = \frac{y'(s)l(x)}{\varphi(x)} \text{ and } 1 = \int_\Omega l(x)dx$$

with l arbitrary. Linear type of initial/boundary conditions are very similar to this. The $\varphi = f$ in the initial/boundary condition is annoying but can be avoided by making the original initial/boundary condition 0 by splitting explained in period1.

Going back because of our choice the following thing can be derived

$$\int_\Omega \varphi(x)L\left(K(t,x)\right)dx = f \Leftrightarrow$$
$$\int_\Omega \varphi(x)\delta(t-x)dx = f \Leftrightarrow$$
$$\varphi(t) = f$$

Boundary Green functions deals with boundary conditions like how source Green functions deals with the source. The intuition behind them is the same as the solutions of a linear homogenous ODE that span a vector space.

$$y(t) = \int_{\partial B} \varphi(x)K(t,x)dx$$

If you impose $L(y(t)) = 0$ on this you get the following:

$$\int_{\partial B} \varphi(x) L(K(t,x)) dx = 0 \Leftarrow$$
$$L(K(t,x)) = 0$$

Again the initial conditions on $K(t,x)$ come from the original problem:
For a dirichlet boundary condition in s the following can be done:

$$y(s) = \int_{\partial \Omega} \varphi(x) K(s,x) dx \Leftarrow$$
$$K(s,x) = \delta(x-s) \text{ and } \varphi(s) = y(s)$$

Other linear type initial/boundary conditions are very similar.
In the Green function method we searched an integral transform with a certain property related to the equation that we were solving. Certain classes of integral transformations have nice properties for a big class of equations. A classic integral transform used for ODEs is the Fourier transform `https://en.wikipedia.org/wiki/Fourier_transform`. But we haven't figured out how to deal with boundary conditions in this case.
The discrete version of an integral transform is a series transform, in which you transfer information about the function to φ_n.

$$y(t) = \sum_{n=0}^{\infty} \varphi_n e_n(t)$$

our original equation with this becomes:

$$L\left(\sum_{n=0}^{\infty} \varphi_n e_n(t)\right) = f \Leftrightarrow$$
$$\sum_{n=0}^{\infty} \varphi_n L(e_n(t)) = f$$

Again there a lot of tricks you can pull of with this. A convenient thing is when $L(e_n(t)) = \phi_n$ (and e_n spans the space of functions which follow the initial/boundary conditions) is bi-orthogonal basis against some ψ_n ($\langle \phi_j \mid \psi_k \rangle = \delta_{jk}$) .

$$\sum_{n=0}^{\infty} \varphi_n \phi_n = f \Rightarrow$$
$$\left\langle \sum_{n=0}^{\infty} \varphi_n \phi_n \mid \psi_j \right\rangle = \langle f \mid \psi_j \rangle \Leftrightarrow$$
$$\sum_{n=0}^{\infty} \varphi_n \langle \phi_n \mid \psi_j \rangle = \langle f \mid \psi_j \rangle \Leftrightarrow$$
$$\sum_{n=0}^{\infty} \varphi_n \delta_{nj} = \langle f \mid \psi_j \rangle \Leftrightarrow$$
$$\varphi_j = \langle f \mid \psi_j \rangle$$

This means we have following expression for the solution:

$$y(t) = \sum_{n=0}^{\infty} \langle f(x) \mid \psi_n(x) \rangle e_n(t) \Leftrightarrow$$
$$y(t) = \left\langle f(x) \mid \sum_{n=0}^{\infty} \psi_n(x) e_n(t) \right\rangle$$

this expression in some cases depending on how the inner product is defined corresponds with the Green function $G(t,x) = \sum_{n=0}^{\infty} \psi_n(x) e_n(t)$.
Related wikipedia page : `https://en.wikipedia.org/wiki/Spectral_theory_of_ordinary_differential_equations`
This is probably not the only way to turn ODEs into integral equations. The Feynman-Kac formula `https://en.wikipedia.org/wiki/Feynman%E2%80%93Kac_formula` is derived in an other way but still kind of obeys the general form we have given.
There are multiple ways to turn problems into integral equations for MC methods but not all those integral equations gives you Monte Carlo methods with the same properties. Things like: the type of the domain chosen, stochastic approximations made, what gets thrown to the source term and what needs to be recursed on determine the properties of the Monte Carlo method obtained.

Each Monte Carlo method doesn't have to be limited to 1 integral equation. One may use a different integral equation for each recursion step made in combination with the different modifications discussed in period1. This makes for a big search space of possible Monte Carlo algorithms.

Another thing to take in consideration is that finding Green functions is difficult a way around that is by throwing everything in the source term but by doing this you probably lose good properties. This approach is taken for example in Grid-Free Monte Carlo for PDEs with Spatially Varying Coefficients.

To test that boundary Green functions work for ODEs we work out following example:

$$y'' = y, y(0) = 1, y'(1) = e$$

with solution $y(t) = e^t$. We chose the splitting Green function method with $\Omega = [-1, 1], f = y$ as discussed as before.

Let's create integral representation of the boundary and source terms separately:

$$y_s'' = f \text{ with } y_s(0) = 0, y_s'(1) = 0$$

And the equation for the corresponding Green function:

$$G''(t,x) = \delta(t-x) \text{ with } G(0,x) = 0, G'(1,x) = 0$$

G must be something continuos piecewise linear with a jump of 1 in the derivative

at $t = x$. By some algebra we find following solution:

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

Write out the solution for $y_s(t)$:

$$y_s(t) = \int_0^1 y(x)G(t,x)dx.$$

We still have some problems to fix with the boundary Green functions ...
The boundary in for ODEs is discrete $\partial\Omega = \{0,1\}$. So instead of an integral we have sum and Green function also splits into 2...

$$y_b'' = 0 \text{ with } y_s(0) = 1, y_s'(1) = e$$

Let's call the boundary function K so that we don't confuse it with G. So we have $K(t,0)$ and $K(t,1)$.The solution for these are easily found:

$$\begin{aligned} &K''(t,0) = 0 \text{ with } K(0,0) = 1, K'(0,1) = 0 \Rightarrow \\ &K(t,0) = 1 \end{aligned}$$

$$\begin{aligned} &K''(t,1) = 0 \text{ with } K(1,0) = 0, K'(1,1) = 1 \Rightarrow \\ &K(t,1) = t \end{aligned}$$

Write out the solution for $y_b(t)$:

$$y_b(t) = 1 + et$$

If you put everything together you get:

$$y(t) = 1 + et + \int_0^1 y(x)G(t,x)dx.$$

In the implementation we use Russian Roulette with probability $1 - \frac{1}{1.3}$ always for stopping. In code this looks like:

Definition 3.1.1 (boundary green function)
The boundary green function of a linear differential problem with linear boundary conditions.

Definition 3.1.2 (source green function)
For

$$L(y(z)) = f.$$

with L a linear differential operator, z a point in the input space of y ,f arbitrary and linear boundary conditions. We define the source green function $G(z,s)$ with following property

$$L(G(z,s)) = \delta(z-s)..$$

and null boundary conditions.

The IVPs that we covered so far were easily transformed into integral equations. This is not the case anymore for BVPs.

Example 3.1.3 ($y'' = y$)
Lets look at following BVP:

$$y'' = y, y(0) = 1, y'(1) = e. \tag{15}$$

with $y(t) = e^t$ as solution. We will be using the green functions of y'' to turn this into an integral equation.

Example 3.1.4 (numerical green functions)
There will be probably some green functions that we need that don't have an analytic expression yet.

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.