Deriving Radiative Backpropagation using the Recursive Form of Path Tracing

Tzu-Mao Li (tzumao@mit.edu)

August 2020

1 Introduction

Recently, Nimier-David et al. proposed a new way to implement differentiable rendering by deriving a new way to compute the gradient using the operator formulation and the property of self-adjoint operators. We show in this article that the same algorithm can be derived using traditional automatic differentiation, through differentiating the recursive form of path tracing. Our article assumes the reader has read Nimier-David et al.'s paper.

Let us look at the following pseudo code of a path tracer (without the next event estimation):

```
def Li(pos, dir_in):
    next_pos = intersect(pos, dir_in)
    bsdf_contrib, dir_out = sample_bsdf(dir_in)
    radiance_contrib = Li(next_pos, dir_out)
    return bsdf_contrib * radiance_contrib
```

For each function in the code, we can apply a transform `f(x)->d_f(x, d_output)`. `d_f` returns `dx` and scatter the adjoints to the referenced global variables (like the scene parameters). This is equivalent to the `adjoint` function in Nimier-David et al.'s paper. We apply this transform recursively to the `Li` function above:

```
def d_Li(pos, dir_in, d_output):
    # "forward pass"
    next_pos = intersect(pos, dir_in)
    bsdf_contrib, dir_out = sample_bsdf(dir_in)
    radiance_contrib = Li(next_pos, dir_out)
    output = bsdf_contrib * radiance_contrib

    # adjoint of output = bsdf_contrib * radiance_contrib
    d_bsdf_contrib = d_output * radiance_contrib
    d_radiance_contrib = d_output * bsdf_contrib
    # adjoint of radiance_contrib = Li(next_pos, dir_out)
    d_next_pos, d_dir_out = d_Li(next_pos, dir_out, d_radiance_contrib)
    # adjoint of bsdf_contrib, dir_out = sample_bsdf(dir_in)
    d_dir_in_bsdf = d_sample_bsdf(dir_in, d_bsdf_contrib, d_dir_out)
    # adjoint of next_pos = intersect(pos, dir_in)
    d_pos, d_dir_in_isect = d_intersect(pos, dir_in, d_next_pos)
    # sum up the two d_dir_in dependencies
    d_dir_in = d_dir_in_bsdf + d_dir_in_isect
    return d_pos, d_dir_in
```

We marked the augmented derivative code in red. Notice the quadratic time complexity due to the recursive `Li` and `d_Li` calls. In practice, the derivatives with respect to the scene parameters are propagated in `d_sample_bsdf` and `d_intersect`, which can be computed using the same machinery. The `d_Li` function above can be easily implemented in a megakernel ray tracing framework such as Optix, using the `raygen` shaders. Extending the code above to handle next event estimation is trivial.

If we compare the code above to Listing 2 in Nimier-David et al.'s paper, we can see that the only difference is that Nimier-David et al. do not propagate the derivative to the PDF of bsdf sampling and the sampling procedure. This difference is due to the different order of importance sampling and differentiation. To see this, given a function $f(x;\theta)$ and an importance sampling function $y = g^{-1}(x;\theta)$ (such that $x = g(y;\theta)$), we can first differentiate the function, then apply the importance sampling reparametrization. Alternatively, we can first apply the importance sampling reparametrization, then differentiate the function:

$$\begin{aligned} &\int \nabla_\theta f(x;\theta) dx \\ =& \int \left(\nabla_\theta f(x;\theta)\right)\big|_{x \to g(y;\theta)} \frac{dx}{dy} dy \\ =& \int \nabla_\theta \left(f(x;\theta)\big|_{x \to g(y;\theta)} \frac{dx}{dy} \right) dy, \end{aligned} \tag{1}$$

where the middle integral corresponds to Nimier-David et al.'s approach (note that the Jacobian $\frac{dx}{dy}$, which is the reciprocal of the PDF of the importance sampler, is not differentiated), and the bottom integral corresponds to our `d_Li`.

2 Memory/Computation Trade-offs

The transformation we applied above is a standard reverse-mode transformation, but why does it not deliver the same result of reverse-mode, that gives the adjoint the same time complexity as the forward computation? The secret lies in the recursive call to `Li` in the adjoint `d_Li`. In standard reverse-mode, the result of the recursive call is supposed to be stored in a tape for later use. The transformation above recompute `Li` redundantly every time we need it in the adjoint radiance. We can fix this by introducing an array to cache the radiance. That is, we apply the technique of memoization. Let us first rewrite the path tracing function:

```
def Li_cached(pos, dir_in, depth = 0, cache = []):
    next_pos = intersect(pos, dir_in)
    bsdf_contrib, dir_out = sample_bsdf(dir_in)
    if len(cache) > depth:
        radiance_contrib = cache[depth]
    else:
        radiance_contrib = Li(next_pos, dir_out, depth + 1, cache)
        cache.append(radiance_contrib)
    return bsdf_contrib * radiance_contrib
```

Then we can derive the adjoint radiance that satisfies the cheap gradient principle:

```
def d_Li_cached(pos, dir_in, d_output, depth = 0, cache = []):
    # "forward pass"
    next_pos = intersect(pos, dir_in)
    bsdf_contrib, dir_out = sample_bsdf(dir_in)
    radiance_contrib = Li_cached(next_pos, dir_out, depth + 1, cache)
    output = bsdf_contrib * radiance_contrib

    # adjoint of output = bsdf_contrib * radiance_contrib
    d_bsdf_contrib = d_output * radiance_contrib
    d_radiance_contrib = d_output * bsdf_contrib
    # adjoint of radiance_contrib = Li_cached(next_pos, dir_out, depth + 1, cache)
    d_next_pos, d_dir_out = d_Li_cached(
        next_pos, dir_out, d_radiance_contrib, depth + 1, cache)
    # adjoint of bsdf_contrib, dir_out = sample_bsdf(dir_in)
    d_dir_in_bsdf = d_sample_bsdf(dir_in, d_bsdf_contrib, d_dir_out)
    # adjoint of next_pos = intersect(pos, dir_in)
    d_pos, d_dir_in_isect = d_intersect(pos, dir_in, d_next_pos)
    # sum up the two d_dir_in dependencies
    d_dir_in = d_dir_in_bsdf + d_dir_in_isect
    return d_pos, d_dir_in
```

Now the new adjoint `d_Li_cached` has the same time complexity as `Li`, since we never recompute the radiance at the same depth. We can choose when to cache the radiance to trade between memory consumption and computation. For example, we can store the radiance only when the depth is a multiple of 10. If we store only $\log(N)$ entries, where N is the maximum depth of the path, then the memory usage is $O(\log(N))$ and the computation time is $O(N \log(N))$.