NLO calculations: Subtraction and Slicing

Alexander Huss

November 12, 2023

Contents

1 Introduction

Let us study a simple toy example for an NLO calculation and investigate what challenges arise in the context of fully differential predictions. We will further have a look at two strategies to implement the IRC calcellation differentially: local subtractions and slicing. These will be contrasted in their respective properties and performance.

2 Inclusive calculation

Let us first set the stage by considering our toy example in an *inclusive* setup, i.e. where we have no restrictions on the phase space whatsoever (think: total cross section). We use dimensional regularization ($D = 4 - 2\epsilon$) to regulate all divergences and will drop all terms $\mathcal{O}(\epsilon)$ in the expressions that follow. We further assume that the Born kinematics is fully constraint such that they do not involve any integration. The only term with a "phase-space" integration is therefore the real contribution, for which we assume a very simple integration domain $x \in [0, 1]$ (the limit of no emission given by $x \to 0$).

With these considerations in place, let's fix the expressions for the virtual ("V") and real-emission ("R") contributions

$$\sigma^{\mathrm{V}}_{\mathrm{inc.}} = \frac{1}{\epsilon} + a\,, \qquad\qquad \sigma^{\mathrm{R}}_{\mathrm{inc.}} = \int_0^1 \mathrm{d}x\, \frac{1+bx}{x^{1+\epsilon}} = -\frac{1}{\epsilon} + b\,, \tag{1}$$

where we have explicilty performed the real-emission integral and dropped terms of $\mathcal{O}(\epsilon)$. We can appreciate how the singularities (poles in ϵ) cancel between the virtual and real corrections (by construction) and the NLO corrections for the inclusive case simply read

$$\sigma^{\delta\mathrm{NLO}}_{\mathrm{inc.}} = \sigma^{\mathrm{V}}_{\mathrm{inc.}} + \sigma^{\mathrm{R}}_{\mathrm{inc.}} = a + b \tag{2}$$

3 Differential predictions

In a differential setup we want to admit arbitraty (IRC safe) measurement functions to act on the phase space. Since in this simple toy example the Born phase space is fully constrained, this means we have to allow for an arbitrary function to appear in the integrand of the real corrections and the virtual piece only gets multiplied by that function in the Born limit, i.e. $x \equiv 0$. In summary, the pieces we need to evaluate are, c.f. Eq. (1),

$$\sigma^{\mathrm{V}}_{\mathrm{diff.}} = \left(\frac{1}{\epsilon} + a\right) \mathcal{J}(0)\,, \qquad\qquad \sigma^{\mathrm{R}}_{\mathrm{diff.}} = \int_0^1 \mathrm{d}x\, \frac{1+bx}{x^{1+\epsilon}}\, \mathcal{J}(x)\,. \tag{3}$$

The main point here is that $\mathcal{J}(x)$ can in general be very complicated (think: isolation algorithms, jet clusterings, energy-energy correlators, ...) such that tackling this integral analytically is inpractical. Moreover, you do not want to re-compute a difficult integral each time the measurement function changes.

From these considerations it becomes apparent that we'll want a flexible *numerical* way of evaluating this integral. The main challenge we then face is that the real-emission integral is actually divergent, which is why we had to introduce a regulator (note that in general, we'll be facing $D = 4 - 2\epsilon$ dimensional integrals, which are non-integer and thus not suitable for a Monte Carlo approach). To expose the singularity, we need to perform some (partial) integration of the real-emission phase space, but at the same time we want to remain fully differential in the kinematics, i.e. want to keep the integration in tact. These seemingly contradicting requirements are what is sloved using so-called *subtraction methods*, which rearrange the divergences so we end up with only well-defined (finite) integrals that we can do numerically.

3.1 Local Subtraction

The local subtraction follows a very common strategy in solving physics problems: find a suitable prarametrisation of a zero. We essentially subtract something differentially and add the same quantity back in integrated form. For this to work, we need a very good understanding of how the singularities arise in our calculations. In real applications

one exploits *factorization* in the IRC limits to devise universal methods to deal with the singularities.

Here, the situation is very simple: the singularity comes from the $x \to 0$ limit of the real-emission integral. So we can find a very simple prescription to isolate the singularity by rewriting the real-emission corrections as follows

$$\begin{aligned}\sigma^{\mathrm{R}}_{\mathrm{diff.}} &= \int_0^1 \mathrm{d}x\, \frac{1+bx}{x^{1+\epsilon}}\, \mathcal{J}(x) \\ &= \int_0^1 \mathrm{d}x\, \frac{1+bx}{x^{1+\epsilon}} \Big[\mathcal{J}(x) - \mathcal{J}(0)\Big] + \mathcal{J}(0) \int_0^1 \mathrm{d}x\, \frac{1+bx}{x^{1+\epsilon}} \\ &= \int_0^1 \mathrm{d}x\, \frac{1+bx}{x} \Big[\mathcal{J}(x) - \mathcal{J}(0)\Big] + \left(-\frac{1}{\epsilon} + b\right) \mathcal{J}(0)\,. \end{aligned} \tag{4}$$

In the last step we set $\epsilon = 0$ in the first term, since it's now finite with the counterterm; the singularity is fully isolated in the second term that we could perform analytically independent of the measurement function.

Combining this with the virtual corrections, we can appreciate how a local subtraction manages to arrange the calculation into finite pieces that are amenable to numerical evaluation:

$$\sigma^{\delta\mathrm{NLO}}_{\mathrm{diff.}} = (a+b)\, \mathcal{J}(0) + \int_0^1 \mathrm{d}x\, \frac{1+bx}{x} \Big[\mathcal{J}(x) - \mathcal{J}(0)\Big] \tag{5}$$

3.1.1 Implementation

The implementation of (5) in a python function is straighforward. We choose the VEGAS algorithm to integrate the real-emission "phase space" and give the number of calls (events) as an argument so we can compare the performance later

```python
def subtr_dNLO(ncall: int) -> tuple[float, float]:

    def integrand(x: list[float]) -> float:
        return (1. + b * x[0]) / x[0] * (J(x[0]) - J(0))

    integ = vegas.Integrator([[0, 1]])
    integ(integrand, nitn=10, neval=(ncall / 10))  # 10% into adaption
    result = integ(integrand, nitn=10, neval=ncall)

    return ((a + b) * J(0) + result.mean, result.sdev)
```

3.2 Non-Local Subtraction: Slicing

In a non-local subtraction, or *slicing* approach, one exploits the fact that the divergence can be isolated with some resolution variable on which we place a cut to regulate the divergence. It is important to note that slicing methods are not "exact" but come with an error that must be controlled and estimated.

In this toy example, we already know that the singularity arises from the $x \to 0$ limit so we can regulate the real-emission integral by imposing $x > \xi > 0$:

$$\begin{aligned}
\sigma^{\mathrm{R}}_{\mathrm{diff.}} &= \int_0^1 \mathrm{d}x \, \frac{1+bx}{x^{1+\epsilon}} \, \mathcal{J}(x) \\
&= \int_0^\xi \mathrm{d}x \, \frac{1+bx}{x^{1+\epsilon}} \, \mathcal{J}(x) + \int_\xi^1 \mathrm{d}x \, \frac{1+bx}{x^{1+\epsilon}} \, \mathcal{J}(x) \\
&= \int_0^\xi \mathrm{d}x \, \frac{1+bx}{x^{1+\epsilon}} \, \Big[\mathcal{J}(0) + \mathcal{O}(\xi^n)\Big] + \int_\xi^1 \mathrm{d}x \, \frac{1+bx}{x} \, \mathcal{J}(x) \\
&= \left(-\frac{1}{\epsilon} + \ln(\xi)\right) \mathcal{J}(0) + \int_\xi^1 \mathrm{d}x \, \frac{1+bx}{x} \, \mathcal{J}(x) + \mathcal{O}(\xi^n) \,. \qquad (6)
\end{aligned}$$

The crucial step is in the third line, where we expanded $\mathcal{J}(x) \simeq \mathcal{J}(0) + \mathcal{O}(\xi^n)$, again allowing to pull out the measurement function out of the integral that diverges. It also becomes clear that there's explicitly an *error* that we make by expanding the measurement function this way and that we'll want to take ξ as small as possible. However, we also see that the regulator induces a $\ln(\xi)$ term that blows up as we take $\xi \to 0$ and we will see how that impacts the performance of the method later. The power n of the error term will depend on the parameter/observable we choose (x) and the measurement function $\mathcal{J}$. Also note that in real-world applications of the method, the "below-cut" contribution ($x \in [0, \xi]$) is typically obtained by expanding a resummation formula for x to the appropriate order. Typically, these expressions are only known to leading power (LP) in x and therefore there is an additional source of power corrections $\sim \xi^m$ from next-to-leading power (NLP) terms that also contributes to the error.

Combining the terms with the virtual corrections, we get the slicing result with the ξ *cutoff* parameter dependence and an associated error

$$\sigma^{\delta\mathrm{NLO}}_{\mathrm{diff.}} = \big(a + \ln(\xi)\big) \, \mathcal{J}(0) + \int_\xi^1 \mathrm{d}x \, \frac{1+bx}{x} \, \mathcal{J}(x) + \mathcal{O}(\xi^n) \qquad (7)$$

3.2.1 Implementation

The implementation of (7) in a python function is analogous to the subtraction case above. This time, we also have the cutoff parameter ξ as an argument as we'll want to vary it and see how the errors change.

```python
def slice_dNLO(xi: float, ncall: int) -> tuple[float, float]:

    if (xi <= 0.) or (xi > 1.):
        raise ValueError("cutoff not in valid range [0,1]: {}".format(xi))

    def integrand(x: list[float]) -> float:
        return (1. + b * x[0]) / x[0] * J(x[0])

    integ = vegas.Integrator([[xi, 1]])
    integ(integrand, nitn=10, neval=(ncall / 10))  # 10% into adaption
    result = integ(integrand, nitn=10, neval=ncall)
```

```
    return ((a + math.log(xi)) * J(0) + result.mean, result.sdev)
```

4 Playground

Let us export the two approaches we introduced above and compare their performance. For simplicity we'll set the constants in the virtual and real corrections to one

```
# the constants in the virtual and real corrections defined globally
a: float = 1.
b: float = 1.
```

For the measurement function, we implement a few to test the dependence of the methods on its properties.

```
# the measurement function (global flag as switch)
iJ: int = 1
def J(x: float) -> float:
    if iJ == 0:
        # total cross section (= a + b)
        return 1.
    elif iJ == 1:
        # linear dependence => linear error term
        return 1. - x
    elif iJ == 2:
        # quadratic dependence & error
        return 1. - x**2
    elif iJ == 3:
        # a more complex one (quadratic)
        return math.cosh(x)
    else:
        raise RuntimeError(
            "unknown measurement function switch: {}".format(iJ))
```

- Feel free to implement your own one here and try it out.

4.1 Export source code

We can export the python source code to a file `main.py`:

```
import math
import vegas
import numpy as np
import sys

# the constants in the virtual and real corrections defined globally
a: float = 1.
b: float = 1.

# the measurement function (global flag as switch)
iJ: int = 1
def J(x: float) -> float:
    if iJ == 0:
```

```
        # total cross section (= a + b)
        return 1.
    elif iJ == 1:
        # linear dependence => linear error term
        return 1. - x
    elif iJ == 2:
        # quadratic dependence & error
        return 1. - x**2
    elif iJ == 3:
        # a more complex one (quadratic)
        return math.cosh(x)
    else:
        raise RuntimeError(
            "unknown measurement function switch: {}".format(iJ))

def subtr_dNLO(ncall: int) -> tuple[float, float]:

    def integrand(x: list[float]) -> float:
        return (1. + b * x[0]) / x[0] * (J(x[0]) - J(0))

    integ = vegas.Integrator([[0, 1]])
    integ(integrand, nitn=10, neval=(ncall / 10))  # 10% into adaption
    result = integ(integrand, nitn=10, neval=ncall)

    return ((a + b) * J(0) + result.mean, result.sdev)

def slice_dNLO(xi: float, ncall: int) -> tuple[float, float]:

    if (xi <= 0.) or (xi > 1.):
        raise ValueError("cutoff not in valid range [0,1]: {}".format(xi))

    def integrand(x: list[float]) -> float:
        return (1. + b * x[0]) / x[0] * J(x[0])

    integ = vegas.Integrator([[xi, 1]])
    integ(integrand, nitn=10, neval=(ncall / 10))  # 10% into adaption
    result = integ(integrand, nitn=10, neval=ncall)

    return ((a + math.log(xi)) * J(0) + result.mean, result.sdev)

if __name__ == "__main__":
    if len(sys.argv) < 3:
        raise RuntimeError("I expect at least two arguments:  [subtr|slice]  ncall")
    ncall = int(sys.argv[2])
    if sys.argv[1].lower() == "subtr":
        res_subtr = subtr_dNLO(ncall=ncall)
        print("{:e} {:e}".format(*res_subtr))
    elif sys.argv[1].lower() == "slice":
        if len(sys.argv) == 6:
            ilow = int(sys.argv[3])
            iupp = int(sys.argv[4])
            nsteps = int(sys.argv[5])
        else:
            ilow = -3
            iupp = -3
            nsteps = 1
        res_subtr = subtr_dNLO(ncall=ncall)
        for xi in np.logspace(ilow, iupp, nsteps):
            res_slice = slice_dNLO(xi=xi, ncall=ncall)
            Del_val = res_slice[0] - res_subtr[0]
            Del_err = math.sqrt(res_slice[1]**2 + res_subtr[1]**2)
            print("{:e}  {:e} {:e}  {:+e} {:e}".format(
```

```
                xi, *res_slice, Del_val, Del_err))
    else:
        raise RuntimeError("unrecognised mode: {}".format(sys.argv[1]))
```

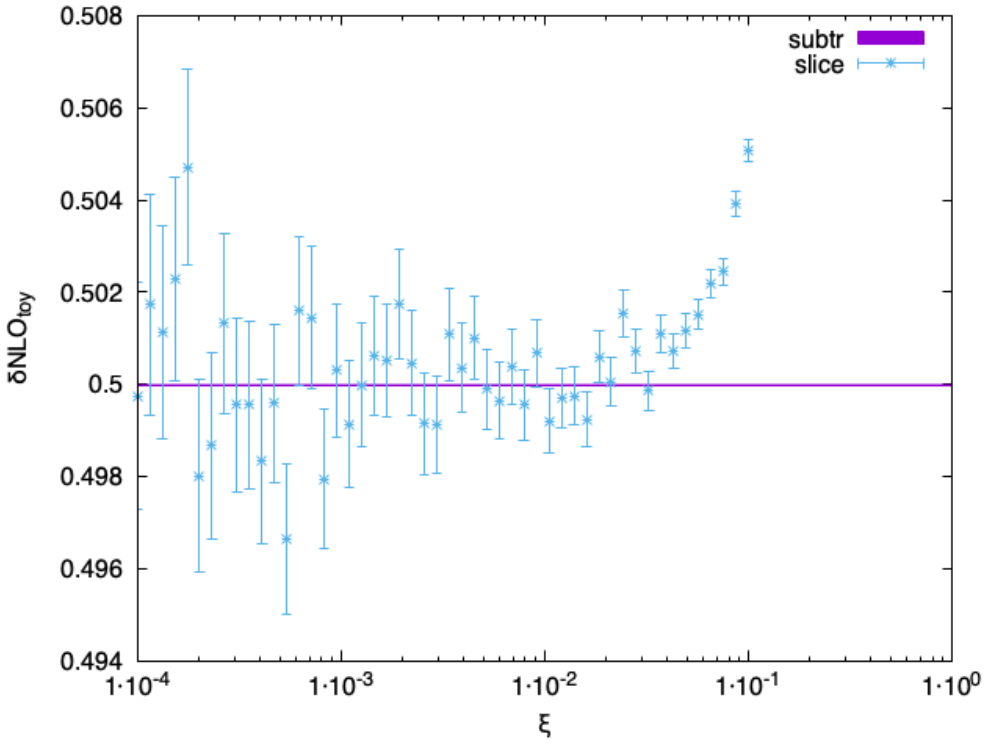
by using the `tangle` command

```
(org-babel-tangle)
```

Let's use the implementation to generate some "events"

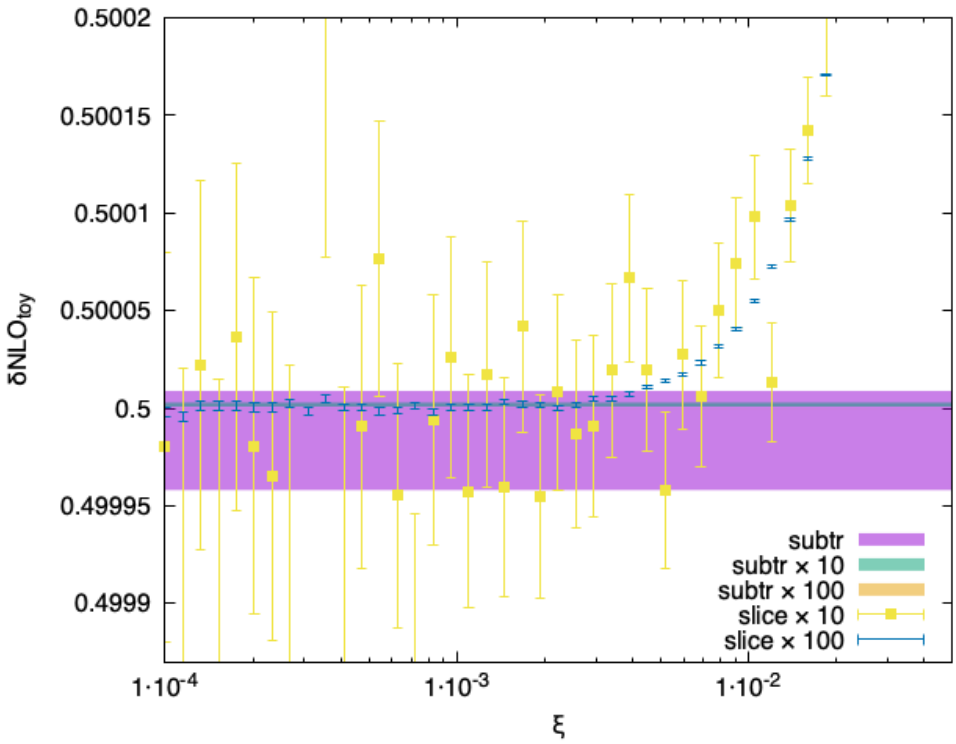
```
python main.py subtr 1000             > data_subtr.dat
python main.py subtr 10000            > data_subtr_10.dat
python main.py subtr 100000           > data_subtr_100.dat
python main.py slice 1000   -4 -1 50 > data_slice.dat
python main.py slice 10000  -4 -1 50 > data_slice_10.dat
python main.py slice 100000 -4 -1 50 > data_slice_100.dat
```

Time to plot the results; let us start with the $N_{\text{call}} = 1000$ case:



We see that the local subtraction yields much smaller uncertainties than the slicing method for the same number of events. We further see the error, which in this default setup is *linear* $\mathcal{O}(\xi)$, increases as we choose larger values of the cutoff. Smaller ξ makes the results less stable; the reason is the large logarithm $\ln(\xi)$ in Eq. (7), which must compensate against the cut-off regulated real-emission part.

From the plot, and the numerical accuracy we have achieved, the result starts to stabilize at around $\xi \simeq 10^{-2}$. However, this is a statement that depends on the accuracy target we have on the final result. To make that point clear, let's look at the data with higher statistics and see how things compare



For this specific example we see that we typically need $\mathcal{O}(10)$ times more statistics in the slicing to achieve a similar numerical precision. Note that in real-life application this can be much worse and also that there's an additional overhead of having to scan the ξ variation to find a "plateau". With the numerical precision of "slice $\times$ 100", we see that the $\mathcal{O}(\xi)$ error only becomes subdominant below $\xi \lesssim 10^{-3}$, that is to note that the smallness of the cutoff ξ is dictated by the target precision we aim for.

- Try to play around with different measurement functions; how does the picture change if the error is quadratic?