Manual for

Version 0.9
correlated emissions only

December 14, 2018

Quick reference guide for `SoftSERVE` 0.9 (correlated emissions only)

Step 0 Install using `./configure` from program root directory (use flag `--MPFR` for multiprecision)

Step 1 Bring measurement function into exponential form (see 4.1)

Step 2 Derive parameters n, m, and functions F_X (4.2,4.3,4.4)

Step 3 Descend into `SCET1` ($n \neq 0$) or `SCET2` ($n = 0$) folder and edit input files (4.5,4.6)

Step 4 Call `make` to compile executables (call `make list` for available individual targets) (4.7)

Step 5 Run binaries from the `Executables` subfolder manually or using `./execsftsrvNAE` script (4.8)

Step 5a Recompile and re-run executables with different `border` setting to check for rounding errors (4.9)

Step 5b Re-build and run executables using multiprecision variables, if needed (4.10)

Step 6 Combine output of all executables manually or using `./sftsrvresNAE` script (4.11)

Step 7 (Only for Fourier space): Call `./fourierconvert` script to account for phases (4.12)

Step 8 Renormalise manually or using `./laprenormNAE` (4.13)

Syntax for the scripts

Script	effect
`./configure`	Install and set up
`./configure --MPFR`	Install with GMP/MPFR support
`make list`	Write list of all possible make targets
`make target`	Build binaries as specified by "target"
`make target BOOST=1`	Build binaries using multiprecision variables
`./execsftsrvNAE -o observable -d subdir`	Run all binaries and tallies results
`./sftsrvresNAE -o observable -d subdir`	Tally results from individual result files
`./fourierconvert -o observable -d subdir`	Creates Fourier-adjusted results file
`./laprenormNAE -o observable -n nvalue -d subdir`	Renormalise in Laplace space, SCET-1
`./laprenormNAE -o observable -d subdir`	Renormalise in Laplace space, SCET-2
`./[any script] -h`	Explain the script

In all cases, the `-d subdir` and `-n nvalue` options (and `-h` flag, of course) are optional.

`target` must be replaced with the colour structure target that is supposed to be built, see `make list`.

`observable`, `nvalue` and `subdir` must be replaced with their observable-dependent values as specified in the input files (`observable` and `n`), or as nested in the directory tree (`subdir`).

Instead of `-o observable`, `-n nvalue`, `-d subdir` and `-h`, the user can use `--obs=observable`, `--n=nvalue`, `--dir=subdir` or `--help`.

Contents

Chapter 1

Overview

This document serves as the manual for users of `SoftSERVE` wishing to compute correlated emission contributions to NNLO bare soft functions for generic observables in settings involving two light-like back-to-back Wilson lines, with subsequent renormalisation. `SoftSERVE` in turn can be seen as the analytic framework laid out in [1] crystallised in C++ code. Therefore [1] should be consulted for any detail on the analytic background in general, and on the criteria deciding whether observables are amenable to computation using `SoftSERVE`, in particular.

This manual is structured to make it accessible mainly for first-time users, with section indices and a dedicated section for known and typical problems to assist troubleshooters and repeat users. To this purpose we will lay out the primary requirements and initial setup for the program suite in chapter2. We then present a brief step-by-step list of the program flow for a typical computation of a hypothetical observable's soft function in chapter 3. Each step on this checklist corresponds and links to a subsection of chapter 4, which explains in detail what needs to be done and why, at that particular step along the way. We also include a list of known problems and difficulties often encountered, and show how to solve or circumvent these in chapter 5. We will add problems encountered by users to this list, it will be updated ad-hoc, as problems are reported to the developers.

Finally, where necessary or helpful we refer to explanatory appendices that lay out the analytic background or computational circumstances that make a given procedure necessary, if these arise due to computational considerations. For features originating in the analytic framework, we refer to the main paper[1] explaining that. We also provide explicit template examples in Appendix E for multiple observables that we have treated with `SoftSERVE`, including the relevant program inputs, which should serve as a guide for users who wish to use `SoftSERVE` to make novel predictions.

Chapter 2

Initial installation and setup

A user seeking to use SoftSERVE needs at base level two things:

- Rudimentary understanding of C++ code, slightly above HelloWorld.cpp level and familiarity with the cmath syntax, and
- a POSIX standard machine equipped with a working C++ compile environment.

The former is required to edit the template C++ files that encode the input for a given observable. For the latter, the program has been tested on several Linux distributions,various MacOS versions, and Cygwin[1] using gcc/g++, clang/clang++ and icc/icpc.

All dependencies are included in the SoftSERVE package. For a vanilla run of SoftSERVE this is essentially the Cuba[2] library of numerical integrators.

To setup SoftSERVE simply expand the contents of the tarball downloaded from HEPForge and run

```
./configure
```

This sets up the compile commands used later on. Running make does not happen at this stage, we use it to make the observable dependent executables later on.

The configuration script contains a short explanation of its features, simply call

```
./configure --help
```

> Attention MacOS users:
>
> If you are only running the Xcode package on your machine and do not have a dedicated, self-contained, gcc installation, you need to run
>
> ```
> ./configure CC=clang CXX=clang++
> ```
>
> instead. If you do not, the configuration script may mistake clang for gcc, and subsequent compilations may fail.

[1] Using Cygwin is explicitly not recommended, and using MinGW will fail, both due to their implementation of fork or lack thereof.

In some cases (see 4.10) the program needs to internally use multiprecision data types, if the typical C++ `double` type variables do not provide enough digits to differentiate between two close-by numbers. For this case we include the header-only `boost`[3] library's implementation of multiprecision variables. This works out of the box and without modification of the installation of `SoftSERVE`.

However, the implementation of multiprecision floating point calculations via `boost` is slow, it is in fact roughly a factor 2 slower than the multiprecision backbone provided by the GMP[4] and MPFR[5] libraries[6]. We therefore ship the GMP and MPFR libraries as well, in case a user really needs the factor of 2.

If so, the GMP and MPFR libraries must be compiled during the `./configure` call, as well. The script should then be called via

```
./configure --MPFR
```

to enforce this.

As GMP is especially prone to miscompilation we do not recommend compiling those libraries unless you find that you really need both the multiprecision capability and the slight increase in performance.

It is possible to reconfigure with `./configure --MPFR` having already called `./configure` alone at some point before, so we highly recommend not including the `--MPFR` flag as the default option.

Actually using GMP or MPFR, rather than Boost, requires a small modification of the source code, see section 4.10 for details.

Chapter 3

Program flow

The actual step-by-step flow through the program is as follows:

Section 4.1 Bring the soft function into the correct, exponential form.

Section 4.2 Apply the parametrisation for the correlated emission case

Section 4.3 Extract the input required for `SoftSERVE` from the correctly parametrised exponential.

Section 4.4 Optional: Check again for compatibility with `SoftSERVE`.

Section 4.5 Write the observable details to the input files.

Section 4.6 Specify the details for the numerical integration.

Section 4.7 Compile the executables.

Section 4.8 Run the executables.

Section 4.9 Check the output for consistency.

Section 4.10 If necessary, recompile using multiprecision and run.

Section 4.11 Optional: Tidy up and get full result.

Section 4.12 Fourier space only: Adjust for phases.

Section 4.13 Optional: Renormalise either by hand or via script.

The reason the approach still requires unscripted action by the user is that maintaining full generality is difficult if the input for the script isn't restricted: We could provide scripts or Mathematica notebooks that handle the parametrisations, but these would likely often fail for intricate observables[1]. We therefore force the user to do the analytic part themselves, which e.g. also allows you to use shortcuts where possible, or use non-trivial analytic properties to simplify expressions.

The next chapter contains all the details about the individual steps phrased as generically as possible. Then, in order to connect to the real world we provide a number of template observables, for both SCET-1 and SCET-2, in Appendix E. These templates describe what the flow of ideas and actions are that leads to the correct input files for each observable treated.

[1] Looking at you, Transverse Thrust...

Chapter 4

Detailed program steps

4.1 Exponential form

The form of the soft function we want to compute, complete to NNLO — two-loop — order, is derived in [1] as

$$\begin{aligned} S(\tau) = 1 &+ \left(\frac{Z_\alpha \alpha_s}{4\pi}\right) \left(\mu^2 \bar{\tau}^2\right)^\epsilon (\nu\bar{\tau})^\alpha \, S_R(\epsilon, \alpha) \\ &+ \left(\frac{Z_\alpha \alpha_s}{4\pi}\right)^2 \left(\mu^2 \bar{\tau}^2\right)^{2\epsilon} \Big((\nu\bar{\tau})^\alpha \, S_{RV}(\epsilon, \alpha) + (\nu\bar{\tau})^{2\alpha} \, S_{RR}(\epsilon, \alpha) \Big) + \mathcal{O}(\alpha_s^3), \end{aligned} \tag{4.1}$$

where the S_i are the Laurent series calculated by `SoftSERVE`. These are of the schematic form

$$S_i \sim \int \underbrace{\left|\mathcal{A}^{(i)}(\{k\})\right|^2}_{\text{Matrix element}} \overbrace{\exp(-\tau\omega(\{k\}))}^{\text{Measurement function}} \underbrace{\prod_{\substack{R:p=k \\ RX:p=k,l}} 2\pi\,\delta^+(p^2) \; \overbrace{(p_+ + p_-)^{-\alpha}}^{\text{Analytic regulator}} \; \frac{\mathrm{d}^d p}{(2\pi)^d}}_{\text{Analytically regularised on-shell phase space}} . \tag{4.2}$$

The matrix elements $\mathcal{A}$ are fixed and the same for all possible dijet soft functions (merely different between loop and real emission numbers) and the analytic regulator is fixed by convention, leaving only the measurement function to be specified.

The constraints on the measurement function are, to recap [1], as follows:

- It is of the given exponential form.
- The variable τ has a unit of inverse energy, demanding that ω, which encodes the information about the observable, has mass dimension one.
- ω observes infrared and collinear safety, meaning in particular that the relations $\omega^{(2)}(k, \alpha k) = \omega^{(1)}([1+\alpha]k))$ and $\omega^{(2)}(k, 0) = \omega^{(1)}(k)$ between its 1- and 2-particle instances hold.
- ω does not depend on the regulators ϵ and α. This can in some cases restrict the transformation to exponential form, if auxiliary Fourier or other transformations are involved.

- ω is real and non-negative in all of phase space (see appendix B of the main paper[1] for purely imaginary cases). Note that this requirement for the numerical treatment is stricter than the assumption for the analytic formulae.
- ω vanishes or diverges only on hypersurfaces of at least codimension one of the full phase space. In other words, ω can vanish or diverge, but the measure of all those points must be zero, it cannot vanish on extended regions of the phase space for the emissions.
- It depends at most on one angle per emission, measured between the transverse space component of the emission momentum and a common reference vector in transverse space.

How this form, in particular the exponential, is achieved, is irrelevant. For many observables, such as event shapes like Thrust or C-Parameter, a Laplace transform will do it.

The Template Guide section at the end of the manual contains examples of several of these "standard" observables (Thrust, Threshold Drell-Yan,...), as well as one purely imaginary Fourier space observable (p_T resummation).

4.2 Correlated emission parametrisation

Now we apply the parametrisation for the correlated emission case (i.e. the summand in the matrix element appearing with $C_f C_A$ or $C_f T_f n_f$ colour factors) to $\omega^{(2)}(k,l)$. For light-cone components of the two emission momenta we apply:

$$\begin{aligned} k_+ &= \frac{b}{a+b}\, p_T \sqrt{y} & \qquad l_+ &= \frac{a}{a+b}\, p_T \sqrt{y} \\ k_- &= \frac{ab}{1+ab}\, p_T \frac{1}{\sqrt{y}} & \qquad l_- &= \frac{1}{1+ab}\, p_T \frac{1}{\sqrt{y}}. \end{aligned} \tag{4.3}$$

We cover the full phase space for k_+, k_-, l_+, and l_- if p_T, a, b, $y \in [0, \infty[$.

The angular parametrisation in the transverse space, assuming the common reference vector is represented by a unit vector $\hat{v}_\perp$, is given by

$$\begin{aligned} \vec{k}_T \cdot \vec{l}_T &= \sqrt{k_+ k_- l_+ l_-}\, \cos\theta_{kl} =: \sqrt{k_+ k_- l_+ l_-}\,(1 - 2t_{kl}) \\ \vec{k}_T \cdot \hat{v}_\perp &= \sqrt{k_+ k_-}\, c_k =: \sqrt{k_+ k_-}\,(1 - 2t_k) \\ \vec{l}_T \cdot \hat{v}_\perp &= \sqrt{l_+ l_-}\, c_l =: \sqrt{l_+ l_-}\,(1 - 2t_l), \end{aligned} \tag{4.4}$$

where the magnitude of the transverse space vectors is set by the on-shell condition, and the coloured variables are coded in the C++ program and can be used as variables in the measurement functions. Whether c_i or t_i are used is left to the preference of the user.

4.3 Extracting input

The only input we need are the functions F_A and F_B, and the the parameters n and m. The one-loop function f is recovered by the program directly via infrared safety.

We extract these from the 2-particle measurement exponential, which now can be written in the form

$$\exp\left(-\tau\omega\left(k,l\right)\right) = \exp\left(-p_T y^{\frac{n}{2}} F(a,b,y,t_{kl},c_k,c_l)\right). \tag{4.5}$$

The p_T dependence factorises because p_T is the only surviving dimensionful quantity, and we factor out $y^{\frac{n}{2}}$ in such a way that F is finite in the limit $y \to 0$[1].

We can now already read off the value for the parameter n, and proceed to constructing the measurement functions F_X.

A quick recap to summarise [1]: The integration runs over domains $[0, \infty[$ for the variables a, b, y, which is bad for numerical integration, which needs finite integral boundaries. We therefore split each integration domain $[1, \infty[$ into two subdomains, $[0, 1]$ and $[1, \infty[$. Combining all possibilities for the different variables we find eight distinct integration regions with different integration boundaries for these 3 variables. One of these, $[0, 1] \times [0, 1] \times [0, 1]$, we identify as region A, and we identify the function F in equation 4.3 as the measurement function F_A.

Symmetry considerations relate the integration results we get from different regions. We find that we only need to consider two subregions, one of which we take to be A, the other one — B — we find via one of three possible mappings:

- Substituting y for $\frac{1}{y}$
- Substituting a for $\frac{1}{a}$
- Substituting b for $\frac{1}{b}$

The measurement function F is in general not the same in the two regions A and B. To get F_B we therefore choose one of the three replacements[2], and apply it to the measurement exponential[3]. We assign the label F_B to the F following the substitution. Note here that the symmetry considerations state that the value of the integrals in all regions from which we could extract F_B match, but not necessarily the relevant integrands. This means that the forms of F_B can depend on which of the three replacements above you choose.

Finally, we pick any of the functions F_X and expand it in the limit of small y, which is of the form $F_X \approx c_0 + y^m c_1 + o(y^m)$, i.e. we extract the parameter m from the first non-constant term in the expansion of F. For most observables we will find $F_X \approx c_0 + yc_1 + \mathcal{O}(y^2)$ (i.e. $m = 1$), but there are also observables for which e.g. $F_X \approx c_0 + \sqrt{y}c_1 + \mathcal{O}(y)$, $(m = 0.5)$, and there may even be other values allowed. If no such value can be extracted, or if the extracted value is zero, or if you're at all uncertain here, use $m = 1$.

4.4 Optional: Checking compatibility

If you're at all uncertain if you've made a mistake, or if your programs threw up error messages that lead you to believe you might have made a mistake in the input extraction. This section lists the constraints as they manifest themselves on the input. You can skip this section if your functions F and parameters m and n look reasonable to you.

If you're unsure, here are the relevant checks:

[1] "Finite", as in "generically finite". There may still be combined limits involving y which vanish/diverge. The usual candidate is $y \to 0$, $c_k, c_l \to 0$. As long as $y \to 0$ alone is not sufficient for a zero/divergence, that's fine.

[2] The choice is yours, which one is the most suitable is observable-dependent. If your observable depends trivially on a and intricately on y, maybe invert a, rather than y.

[3] Careful: The substitutions involving a and b can be applied to F_A directly, the substitution for y must also be applied to the factor $y^{\frac{n}{2}}$.

- We should now have for the correlated emission input:
  - The value for n
  - The two functions[4] $F_X(a, b, y, t, c_k, c_l)$ with $X \in \{A, B\}$
  - The value for m, except for rare cases, where we use $m = 1$.
- Both functions F must be non-negative in all, and finite and non-vanishing in most of phase space, meaning the regions b, a, y, $t \in [0, 1]$, and c_k, $c_l \in [-1, 1]$ or t_k, $t_l \in [0, 1]$. Here "most" of phase space means that they vanish or diverge only on surfaces of codimension ≥ 1. As you will need it in the next step, it is advisable to make a list of all appearing zeroes and divergences here already.
- Any combination of one or more of the limits $y \to 0$, $b \to 0$, as well as the combined limit $(a, t) \to (1, 0)$ must be finite. Only when other variables take on special values alongside them may the functions F vanish or diverge. If this constraint is violated you either made a mistake in the derivation of the functions F, or your observable is not infrared and collinear safe.
- The limits $b \to 0$ and $(a, t, c_k) \to (1, 0, c_l)$ of the functions F must reduce to the same function[5], up to exchange of indices k and l.

4.5 Writing input files

Armed with the functions F and the parameters n and m we can now adjust the input files as needed.

First, we choose the branch of the program to be used based on the value for n. If $n \neq 0$, the correct branch is the SCET-1 branch. If $n = 0$, we must use the SCET-2 branch. We descend into the appropriate subdirectory of the main `SoftSERVE` package, and everything that follows is applicable to both cases identically.

4.5.1 `Input_Common.cpp` and `Input_Parameters.h`

We start with the input that is common for all colour structures, in `Input_Common.cpp`.

Here, we give the observable project a name by assigning it to the string called `observable`. The default is set to "Observable".

We then turn to parametric dependence in the observable. Many observables depend on parameters, e.g. the Angularities event shape with its parameter A, or Transverse Thrust, which depends on a non-dynamical angle between jet- and beam-axes. These parameters must be properly declared and defined in the program, as they can't be left parametric for a numerical evaluation.

To that end we need to add a line of the form

```
datatype name=value;
```

[4] c_k and c_l may be resolved in terms of t_k and t_l.

[5] As a further consistency check you can check if this function matches the function f extracted from the 1-particle measurement exponential if you parametrise the emission momentum l as $l_+ = p_T\sqrt{y}$, $l_- = p_T\frac{1}{\sqrt{y}}$, $\vec{l}_T \cdot \vec{n}_j = p_T c_l$, and write the exponential as $\exp(-\tau\omega(k)) = \exp(-\tau p_T y^{\frac{n}{2}} f(y, c_l))$

to the `Input_Common.cpp` file for a "datatype"-type parameter with name "name" and value "value", and to make it visible to the rest of the program we add the line

```
extern datatype name;
```

to the `Input_Parameters.h` file. Both locations are marked with appropriate comments in the respective files, and the default template does declare and define a floating point parameter called `placeholder`, and sets its value to 0.5.

The usual datatype for parameters will be `double`[6], and the name can be freely chosen, with the exception of a few names which are already taken by internal functions and variables of the program. A list of prohibited names are listed in table 4.1. This list is not complete, so if your code fails to compile, try giving the parameters a different name.

```
a    A    B    beta    c    ck    Ck    Ck1    Ck2    cl    Cl    decimal
f    FA    FB    FA1    FA2    FB1    FB2    GAa    GAb    GA1a    GA1b    GA2a
GA2b    GB    GB1    GB2
n    m    M    mexp    mm    q    r    s    s5    s6    t    T    t5    tk    Tk1
Tk2    tkl    t6    u    v    w    x    xk    xl    y    yk    yl    z
```

Table 4.1: List of prohibited parameter names. A large number of longer names of functions has been omitted, as we judge the chance of your parameters being called `maxpass` or `PreRVEM3` rather low.

> Note: This list only applies to names of parameters you define for your observable. In the `Input` files you should of course use `a`, `y`, and all other variables directly — you can't use them as parameter names precisely because they're already taken by the integration variables.

Finally for `Input_Common.cpp` we input the value for n (SCET-1 case only) and m extracted before at the appropriate places[7].

4.5.2 `Input_Measurement_Correlated.cpp`

Here we insert the definitions of the functions F_X we derived using the correlated emission parametrisation below the giant comment stating "Measurement functions Correlated". There are two predefined C++ functions, one for each of the two functions, of the form (here for A)

```
double FA(v,t6,y,tk,u,b) {
((variable definitions))

   ((comment))
   FA=[[assignment]];
   [[catching zeroes or divergences]]
```

[6]Though `integer` and `float` are also possible.

[7]Reminder: $m \overset{!}{>} 0$, and if that doesn't come out naturally, use $m = 1$.

```
   ((fail-safe and return))
}
```

where double brackets denote a code fragment by its purpose, and round vs. square brackets denote pre- vs. user-defined input. The F_X functions must be entered at `[[assignment]]` in C++ `cmath` syntax, and all zeroes or divergences must be set to a value other than 0 or ∞ in `[[catching zeroes or divergences]]`. This also applies to non-trivial limits of e.g. the form $\frac{a}{a+b}$, which is analytically finite for all relevant variable values, but can look to a straightforward evaluation like the ill-defined $\frac{0}{0}$. It is advisable to treat this as an exercise in evading Murphy's Law: If there is any possible way for your function to be evaluated that results in superficially ambiguous expressions, C++ will almost certainly find it. Here we try to idiot-proof the input accordingly[8], via if-clauses that set the function to its correct value in these cases. There are a few template observables that show this problem.

The template code assumes a measurement function $F_X = \sqrt{a}(1 + \text{placeholder} * c_k)^2$, which is zero at $a = 0$. Accordingly, this zero is defused using an `if`-clause in `[[catching zeroes or divergences]]`. We suggest setting the function to 1 in such cases.

For the definition of angle dependent functions you can use either the predefined `ck` and `cl` as stand-ins for $\cos\theta_k$ and $\cos\theta_l$, or `tk` or `tl` for the t_i defined by $\cos\theta_i = 1 - 2t_i$, or you can mix and match.

> If you feel the need to define internal variables to store intermediate results inside one of the functions F_X, please use `decimal` type variables wherever you'd usually use `double` type variables. This is related to multiprecision issues and is explained in appendix C.

For an argument about why setting arbitrary function values at zeroes is okay, as well as more information about the parameter m, which is related to this issue, see appendix A.

4.6 `Input_Integrator.cpp`

Now we proceed to the settings for the integrator, where we here list all settings and how they can be adjusted.

First, note that some setting variables come with integer suffixes, like `epsrel2` or `key12`. This originates from a splitting in the number of integration variables, with the reasoning that lower dimensional integrals are easier to solve and will converge quicker to more precise values. So we can increase the precision of the overall result at fixed running time by computing the lower dimensional integrals really precisely, but allowing the higher dimensional ones to be a bit less precise, and having them all finish in reasonable time.

We find that we need between 2 and 5 integration dimensions, so we have variables like `epsrelI` with `I`$\in \{2, 3, 4, 5\}$.

The relevant settings now are:

`epsrelI` The required relative accuracy, i.e. we demand that the numerical evaluation for the `I`-dimensional integral terminates only once, for a given integral estimate E and error estimate

[8] In the words of The Doctor: 'The trouble with computers is, of course, that they're very sophisticated idiots."

Δ, the relation $\Delta \leq |\texttt{epsrelI} \cdot E|$ is fulfilled. Due to technical reasons the accuracy we achieve is typically of one to two orders of magnitude higher than the what is set here. Diminishing returns set in rather quickly — varying the default settings by more than two orders of magnitude will usually only increase computing time, not the precision.

The default usually produces results accurate to $\sim 10^{-3}$ relative precision. A second set of suggested variables for $\sim 10^{-5}$ relative precision is provided, as well.

`keyJI` Although the Cuba library[2] provides several numerical integrators, we only use one: the choice of *Divonne* is hard-coded. The three settings `key1I`, `key2I`, and `key3I` set the precise integration strategy *Divonne* uses for the `I`-dimensional integration, i.e. whether to use random number sampling or cubature rules, the choice of random number generator, etc.

For details we refer to the Cuba manual [2]. The default settings amount to a first partitioning stage using deterministic cubature rules, followed by a sampling stage using Korobov random numbers, followed by a refinement strategy.

We strongly suggest not changing these settings. Trial and error has shown that this combination of partitioning and sampling yields the best results - it is the origin of the two order of magnitude improvement we mention above; Korobov sampling improves on cubature. Other settings don't show this feature, or may even increase the error estimate in the sampling step.

`maxpassI` *Divonne*'s partition stage as a crude first approximation may miss important feature of the integrand. We therefore can force it to add an additional `maxpass` partitioning steps once it has concluded that it is ready to proceed to the sampling stage. This is the easiest dial to crank up precision, and the only difference between the standard and precision template settings is here.

This value is essentially arbitrary, but 25 yields good results.

`maxchisq` and the next setting,

`mindev,` set the criteria for the refinement stage. *Divonne* partitions the integration region, and then samples. If the tentative result for a partition region from the partitioning stage, when compared to the final result from sampling, fails a χ^2-test by exceeding `maxchisq`, the subregion is sent to refinement if the error estimate of its contribution to the total error estimate is at least a fraction of `mindev` of the total error estimate. In other words, *Divonne* only refines if there is both something to refine and doing so is worth it.

The default values work well enough here. Play around with them at your leisure.

`border` This setting is the reason we hard-coded *Divonne*. It delineates an exclusion zone of width `border` at the boundaries of the integration region. Sampling points inside this region are not sampled directly, but extrapolated from two points outside the exclusion zone, i.e. inside the bulk of the integration region. We use this feature because the integrand exhibits suppressed logarithmic divergences of the form $x \ln x$ at the integration boundaries. Using the `border` feature suppresses numerical instabilities. The full explanation can be found in appendix A.

We suggest varying the default value by two orders of magnitude at best — if it is too small we encounter numerical instabilities, and if it is too large we introduce systematic uncertainties. It is also related to the "`BOOST`" feature, which we encounter in section 4.10.

`maxevalI` This feature introduces a hard cutoff on the number of integrand function evaluations. We want in general to be guided by the error estimates, hence we choose a large value for these variables. Nevertheless, in some cases it can be nice to have a setting that just kills the integration if it takes too long, so we leave this here.

`seed` In case the integration strategy is changed (different `key` settings), `seed` governs the choice of random number generator. For details see the Cuba manual[2].

`flags` This variable sets the verbosity of the integrator. `flags=3` floods you with log files, whereas `flags=0` is silent. Choose intermediate integers freely.

`epsabsI` Purely maintained for compatibility. Where `epsrelI` sets a termination constraint on the relative error, `epsabsI` sets one on the absolute error. Whichever is fulfilled first, wins. For the numerical integration we separate constant prefactors from integrand structures, so `epsabsI` would set constraints on meaningless intermediate results, so we don't use it. Listed purely in case somebody really needs it, for whatever reason.

4.7 Compiling executables

Having edited the input files we are now ready to compile and then run the executables for a given observable's soft function.

To that end we call `make` from the folder containing the input files and the makefile.

The makefile will compile the different object files containing numerator functions, prefactors and the input data, and link everything together to form three files in the `Executables` folder. The filenames are generated from the value of the `observable` string in `Input_Common.cpp`, by adding one of three suffixes:

`1P` This executable will calculate the regulator pole coefficients for the 1-loop soft function and the 2-loop real-virtual interference contribution, to the orders required for renormalisation[9]

`CA` denotes the executable for the $C_F C_A$ colour structure, absent the real-virtual interference 1-particle cut contribution, which is computed by the `1P` executable

`NF` The $C_F T_f n_F$ contribution to the bare soft function is computed by the executable with this suffix.

The strategy using `make` is sensitive to recycling of object files. After the first run the input-independent object files do not have to be recompiled, and only files modified between `make` calls will be recompiled.

To single out individual colour structures for compilations and to provide auxiliary functions, the following targets have been defined:

`all` Default, same as running `make` without a target

`list` Prints a short card listing all targets and their function

`clean` Removes the intermediate object files

`purge` Removes the intermediate object files and all executables in the `Executables` folder adhering to the standard naming scheme (i.e. ending in one of the three suffixes). Subfolders of `Executables` are not affected.

[9] i.e. there are positive regulator power coefficients computed in the 1-loop result.

`1P`	Only compiles the `1P` executable
`CA`	Only compiles the `CA` executable
`NF`	Only compiles the `NF` executable
`correlated`	Only compiles the `1P`, `CA` and `NF` executables

To use the `boost` library's multiprecision variables, which is described in detail in section 4.10, we set[10] `BOOST=1`.

So to compile the $C_F C_A$ colour structure executables using multiprecision variables we'd call

```
make CA BOOST=1
```

Without multiprecision variables, we just use

```
make CA
```

4.8 Running the executables

To run the executable simply either descend into the `Executables` folder and run them directly and manually, or run the `execsftsrvNAE` script from the directory containing the input files and makefile. Its syntax is

```
./execsftsrvNAE -o observable -d subdir
```

where `observable` must match the name of the binaries without suffix[11], and `subdir` is an optional argument specifying a subdirectory in the `Executables` folder, in which the binaries may reside.

The `./execsftsrvNAE` script runs the three executables in the order 1P $\rightarrow$ NF $\rightarrow$ CA, which corresponds for typical observables to the ordering "fastest-to-slowest", meaning that `1P` usually terminates quicker than `NF`, with `CA` taking the longest. It then combines the three results into one final result, writes that to the file "`Result observable Full.txt`", and moves the executables to a subfolder for storage.

The executables will run sequential numerical integrations over 2D (first) to 5D (last) domains. This means that for a given colour structure the leading, most divergent, regulator poles are the first results to be available. This strategy was chosen because mistakes in the input can then in some cases be spotted from the leading poles, which are available quickly.

During the executables' run the status and results are printed to the console, the results are also written to a result file, and the output of the numerical integration is funnelled into log files, with result and log files both in the resident folder of the executable that's running.

4.9 Consistency checks

So you've run the executables, your integrations have terminated and the results are in. Now we need to check for the single biggest problem facing our program: Rounding errors.

[10] In principle any value different from 0 is fine, even strings. `BOOST=thecakeisalie` would work as well.

[11] i.e. if `observable` is "`abcd`", the script expects to find three binaries called "`abcd 1P`", "`abcd CA`", and "`abcd NF`"

The details of why this problem appears can be found in appendix 4.10, so here we only mention how to check for its appearance. Put succinctly the problem is related to the fact that `double` type variables in C++ only store about ~ 15 digits of a number, which can lead to problems if two almost identical numbers are subtracted. In most cases the appearance of the problem will manifest itself by the numerical evaluation returning `NaN`, but in principle it can lead to simply a wrong result which is not obviously wrong at first glance.

Fortunately for us the problem only appears in its critical form at the boundaries of the integration domain, which is subject to special treatment thanks to the `border`[12] variable.

To check for rounding errors we therefore suggest that you run the program at least twice, the second time with a settings for `border` which differs from the first by two orders of magnitude. 10^{-6} and 10^{-8} are good values, for example. If you have read appendix C and are confident that your observable is not affected, you can skip this step, at your own peril.

If you do run twice and do get different results between the two runs, or if one of the two runs returns `NaN`, follow the strategy in the next section, and see if that solves the problem. If the result is the same with different values for `border`, feel free to proceed immediately to the renormalisation in section 4.13.

4.10 Multiprecision

So your program either `NaN`ed or returns `border`-dependent values. The solution for this problem is to use internal variable types which resolve more digits than the 15 present in `double`. The simplest way of doing that is to only change the command used to compile the executables in section 4.7 and add `BOOST=1` to the `make` command. The files you then generate use multiprecision variables and should not be affected anymore. However, they will be significantly slower than "standard" files, so it is advisable to lower the requested accuracy in the `epsrel` variables.

In some cases the programs are unacceptably slow, or the accuracy cannot be lowered significantly. In this case there is not much that can be done, but we can tease an additional factor of ~ 2 in run time out of the code, by using a different implementation of the multiprecision variables.

The standard implementation is provided by the `boost` library, which is a header-only library and therefore does not need to be compiled. The alternative implementations are provided by the GMP[4] and MPFR[5] libraries, which do need to be compiled. To do that, we ascend to the `SoftSERVE` main directory, and call the configurations script with the appropriate flag:

```
./configure --MPFR
```

This compiles the GMP and MPFR libraries, and writes the output of their configuration and compilation steps to the `GMPcompilation.log` and `MPFRcompilation.log` files. Once the configuration script has terminated, check these for potential errors.

Assuming GMP and MPFR have compiled correctly, you now need to choose the multiprecision provider you want to use. The standard is still the `boost` implementation, even if GMP and MPFR are now compiled. To change this open the problematic observable's `Input_Parameters.h`[13], and look for the line

[12] See section 4.6

[13] Note that this means the setting is tied to one observable. This is not a global change on the level of the full framework.

```
#define BOOST_H
```

As per the comment preceding it we know what we want to do, namely we change it to

```
#define GMP_H
```

if we want to use the GMP library, or

```
#define MPFR_H
```

for the MPFR implementation.

If you now call the makefile with the `BOOST=1` setting, the executables are compiled using `boost`-, GMP- or MPFR-provided multiprecision variables, according to what you `#define`d

The executables compiled using `make` *without* setting `BOOST=1` are unaffected by all of this.

Now compute the bare soft functions again using the newly compiled executables and perform the consistency check again. If it still fails, something else is going on, so please check the troubleshooting chapter 5 in this case.

4.11 Tidying up and getting the full result

Once you are convinced that the result is correct and the integrations are finished, you find the results in the `Result observable XX.txt` files, where `observable` is replaced with the value of the `observable` string in the input files, and `XX` is the executable in question, i.e. `1P`, `CA` and `NF`.

To save you the trouble of having to perform the sum of `CA` and the real-virtual correction from `1P` for the full $C_F C_A$ structure — and in particular to save you the trouble of having to combine the error estimates —, we provide a script to do that for you. Once all result files for a given observable are in, you can call

```
./sftsrvresNAE -o observable -d subdir
```

where `observable` is the value of the `observable` string in the input files, surrounded by quotes if it contains blanks, and `subdir` is the name of the subdirectory of the `Executables` folder that contains the three individual results files. The latter argument is optional and can be omitted, if the files are in `Executables` directly. `./sftsrvresNAE -h`. or `./sftsrvresNAE --help` provides help and an example.

The `./execsftsrvNAE` script calls this script automatically, and the final result is written to the file `Result observable Full.txt`, again with `observable` specified in the input.

4.12 Fourier-adjusting the results (Fourier space functions only)

> If you're wondering if you have to follow this section's instructions, you very likely don't. In this case, feel free to skip ahead to the renormalisation section.
>
> If you have read the relevant appendix in the main paper and are calculating soft functions involving imaginary measurement functions, this is for you.

In an appendix of the main paper we outline how we can compute the real part of Fourier space soft functions using `SoftSERVE`, by running it on the absolute value of the measurement function, and subsequently multiplying the result with $\cos\frac{(2\epsilon+\alpha)\pi}{2}$, $\cos\frac{(4\epsilon+\alpha)\pi}{2}$, or $\cos(2\epsilon+\alpha)\pi$, for 1-loop, 2-loop interference, and 2-loop 2-particle results, respectively[14].

To spare the user the effort of having to do this multiplication manually we provide a script to do it, instead. The `./fourierconvert` script requires the presence of the results files for the `SoftSERVE` run mentioned above (either combined or a full set of individual results files), and is called via `./fourierconvert - o observable -d subdir`, with `observable` the name of the observable, as provided to the `SoftSERVE` input (and as appearing in the names of the results files, mainly), and `subdir` the optional name of the subdirectory of the `Executables` folder, in which the results files reside.

The script performs the required linear combinations of existing regulator orders with the relevant powers of π coefficients (the cosine expanded), and calculates the modified error bars via the sum of squares. It then writes the adjusted results and error bars to a file mirroring the structure of `SoftSERVE` results files for an observable called `Fourier_observable`, where `observable` was the name of the original input.

So for an observable called `whatevs` renormalising multiplicatively in Fourier space, with (imaginary) measurement function iF, the `SoftSERVE` sequence would be

1. Write input files using the measurement function $|F|$
2. Compile the binaries
3. Call `./execsftsrv -o whatevs` to run the integrations
4. Call `./fourierconvert -o whatevs` to adjust the results and account for the cosine factors
5. Call `./laprenorm -o Fourier_whatevs` to renormalise

4.13 Renormalising

Having derived the full bare soft function we now want to renormalise. In principle we should now undo the transformations we performed in section 4.1 to arrive at the soft function in its original space, and then renormalise there. We leave this task for complicated transformations to the user.

However, there are many observables renormalising multiplicatively in the space in which the `SoftSERVE` calculation occurs (usually Laplace or Fourier space), for which we have scripted the renormalisation procedure. With "multiplicatively" we mean that the bare and renormalised soft functions are related by multiplication with a Z-factor, rather than a convolution with it.

Assuming we have a full set of result files for the individual executables in section 4.8, or a combined result file generated via script 4.11, we can call the `./laprenormNAE` script to renormalise such observables.

The script uses the syntax `./laprenormNAE -o observable -d subdirectory -n nvalue` for the SCET-1 case, or `./laprenormNAE -o observable -d subdirectory` for the SCET-2 case, and extract the anomalous dimensions and matching corrections (SCET-1), or anomaly exponents (SCET-2) following the conventions used in [1], especially the "Renormalisation" and "Numerical implementation" sections.

[14] With $\alpha = 0$ in the SCET-1 case, of course.

`observable` must be replaced with the string in the `observable` variable of the input files, and the `subdirectory` and `nvalue` options are optional, specifying a subfolder of the `Executables` folder which contains the result files on which the script operates, and the value of the parameter n, respectively. The `-n` option only exists in case the script cannot read the value from the results files, in which case you'll be asked to provide it manually via this option. If either `observable` or `subdirectory` contain blank spaces, they must be enclosed by single or double quotes, or escaped using backslashes.

In both cases the renormalisation script requires either a set of all three individual result files, or one combined result file as output by the programs and scripts described in the sections above.

All scripts come with the flags `--help` and `-h`, which list their features again.

Examples for observables renormalising in calculation space are (Laplace space) event shapes like C-Parameter or Angularities, as well as observables like Drell-Yan production at threshold, or weak boson production at large transverse momentum, and (Fourier space) transverse momentum resummation.

Chapter 5

Troubleshooting and tips and tricks

This list will be periodically updated as more 'trouble to be shot' is reported to the developers.

5.1 General grievances

- GMP and MPFR will fail to compile if the absolute `SoftSERVE` directory path contains spaces (programs using autotools generally have this problem).
- Functions exhibiting zeroes or divregences in the bulk of the integration region (e.g. p_T resummation) will yield results at reduced accuracy. This is because the zero/divergence generates a logarithmic divergence in the numerical integrand, which introduces instabilities. Such observables can also produce `NaN` as an integration result if the accuracy goal is chosen too ambitiously. This is essentially the numerical integrator giving up.
- Results for the higher reguator orders tend to appear with an insecure error estimate ("Probability of incorrect error estimate: 1"). This warning tends to be overly paranoid, mainly because these results consist of multiple terms originating from multiple integrations that are added up for the final result. `SoftSERVE` estimates the probability that the final result has incorrect error bars as the probability that at least one contributing intermediate result has incorrect error bars, which is the most conservative way to estimate this. Increasing the accuracy can get rid of this warning, but in many cases it can be ignored, if there are no other warning signs (like extensive refining required in the main integration part of the numerical integration, as seen by the appearance of "Split" in the log files, which indicates a complicated structure of the integrand not resolved by the partitioning stage), as the error bars of lower dimensional integrations tend to be very conservative (For known literature results the error bars tend to overestimate the actual deviation by up to two orders of magnitude) and partially make up for the higher dimensional ones.
- Renormalising a Fourier space observable using the `./laprenorm` script still mentions "Laplace space renormalised [...]". This can be ignored, the script was written for Laplace space, it just can be used for Fourier space by sheer accident.

5.2 Cryptic compilation and run time error messages and their meaning

- Cuba 4.2 throws up compiler warnings (can be found in the logfile) warning that vanilla C does not like floating point variables in some places, that some sources are deprecated, and that some libraries are built in deterministic mode. These are not a problem, the libraries compile just fine.

5.3 Tips and tricks

- Executables for machines other than the one used to compile them can be produced by statically linking against the libraries. To do so set `CXXFLAGS=-static` during the `make` call.
- The number of cores the Cuba library uses is dynamically determined by the number of idle cores. If you want to force Cuba to use a fixed number of cores, set the environment variable `CUBACORES` to however many cores you want to use.

Chapter 6

License and contact details

This program uses parts of the `boost`, and GMP and MPFR libraries, which are released under the Boost Software License, dual Lesser GNU Public Licence v3 and GPLv2, and LGPLv3, respectively.

We therefore release `SoftSERVE` under the terms of the GPLv3. Its full text can be found in the `COPYING` file in the `SoftSERVE` main directory.

The developers of `SoftSERVE` are Guido Bell, Rudi Rahn and Jim Talbert. You can contact us at softserve@projects.hepforge.org. Emails sent to this address are forwarded to each of us.

Appendix A

Endpoint suppression

Here we look at some transformations that can improve the convergence rate of the numerical integration, and simplify the behaviour at the boundaries of the integration region.

All of these transformations and substitutions are handled internally by the program and are mostly invisible to the user. However, in some cases knowledge of internal mechanisms is important and influences how the user should operate. We therefore explain these transformations here.

A.1 Non-critical boundaries

The problem we try to solve appears due to the presence of structures like $[t(1-t)]^{-\frac{1}{2}-\epsilon}$ from the angular parametrisation and $b^{-2\epsilon}$ from the regularised phase space measure[1]. We have to expand to subleading orders in ϵ, which means that we find square root and logarithmic divergences in the integrand. These are integrable divergences, but they potentially pose a problem for a numerical integration, which samples points in the integration region and therefore should ideally not have to sample the vicinity of an unbounded peak.

Moreover, the presence of square root divergences is incompatible with numerical integration routines that rely on variance reduction techniques, as *Divonne* and the Cuba library in general do. This is due to the fact a numerical integrator essentially computes an estimate for the mean of the integrand function probabilistically, and uses that the variance of this estimate (which serves as a good error estimate) is determined by the variance of the integrand function and the sample size. The variance of the integrand function is usually not known, but can be estimated from the sample variance.

The problem is now that for square root divergent integrand functions the variance of the integrand is proportional to $\int \mathrm{d}x\ f(x)^2 \sim \int \mathrm{d}x\ \frac{1}{x}$, which is infinite. The sample variance on the other hand is always finite. So an adaptive integration technique which seeks to reduce the variance to reduce the error estimates and to improve the integration precision will fail, because it operates under the wrong assumption that the variance of the integrand function is finite[2].

[1] The latter appears only in the $C_F T_f n_f$ structure, the former appears in all cases.

[2] As a side note: The *Vegas* algorithm is less susceptible to this problem, but *Divonne* is very sensitive to it.

Fortunately we can solve this problem by creatively substituting: We substitute for purely logarithmic divergences at 0:

$$\int_0^1 \mathrm{d}b\, b^{-2\epsilon} X(b) = \int_0^1 \mathrm{d}c^2\, c^{-4\epsilon} X(c^2) = 2\int_0^1 \mathrm{d}c\, c^{1-4\epsilon} X(c^2) \approx 2\int_0^1 \mathrm{d}c\, X(c^2)(c - 4\epsilon\, c \ln c + \mathcal{O}(\epsilon^2)), \quad \text{(A.1)}$$

where we take X to be a finite function containing all other factors in our problem, and have made explicit that the logarithmic divergences that appear in the regulator expansion are now suppressed and of the form $c \ln^n c$.

We can extend this to square root divergences (here at v=0) by substituting to higher powers:

$$\int_0^1 \mathrm{d}v\, v^{-\frac{1}{2}-\epsilon} X(v) = \int_0^1 \mathrm{d}w^4\, w^{-2-4\epsilon} X(w^4) = 4\int_0^1 \mathrm{d}w\, w^{1-4\epsilon} X(w^4) \approx 4\int_0^1 \mathrm{d}w\, X(w^4)(w - 4\epsilon\, w \ln w + \ldots \quad \text{(A.2)}$$

Finally, we can even deal with divergences at 0 and 1, by substituting

$$t = 1 - (1 - s^i)^j \quad \text{(A.3)}$$

This scales as js^i for small s, and $i(1-s)^j$ for small $1-s$, and can therefore suppress appropriate divergences at both ends. As an example, if there is a logarithmic divergence at 0 and a square root divergence at 1, we'd choose $i = 2$ and $j = 4$, for square root divergences at both ends we'd use $i = j = 4$.

The integrand functions we use are full of such divergences, we therefore suppress most limits that are not associated with a divergence.

A.2 Consequences for the measurement function

The endpoint suppression has interesting consequences for the measurement function.

As can be found in [1], the measurement functions F only appear as $F^{4\epsilon}$ in the integrand, which means that at higher orders in ϵ they contribute at best integrable logarithmic divergences. If now such a zero/divergence for F coincides with a suppressed limit, we never have to worry about the divergence, and we never have to worry about $\ln F = \ln 0$ not being defined: The logarithm is suppressed and of the form $0 \ln 0$. In all those cases we can therefore include in the input C++ files an `if`-clause which sets the measurement function to any positive number other than zero.

This is the reasoning behind many of the clauses in the template observables: If the observable vanishes at the combination $a = 0$, $b = \sqrt{e}^{\pi}$, we do not have to worry about the value for b in the combined limit, as $a = 0$ is always suppressed.

In table A.1 we list all the limits which are suppressed, and some which are not suppressed. For the typical user, the measurement function will vanish or diverge in some limits. If the limit in which it is zero/divergent is one of the suppressed limits above, the measurement function at the zero/divergence[3] can be reset to a harmless value without changing the integral at all.

Critical limits on their own can never lead to a vanishing or divergent measurement function, based on infrared and collinear safety, but they have their own way of creating trouble. We therefore list them and employ a special suppression technique for them as well, as will be explained in section A.3.

[3] which `SoftSERVE` would complain about by throwing up error messages

Limit	divergent	suppressed
$b \to 0$	logarithmic ($C_F T_f n_f$)	yes
	critical ($C_F C_A$)	yes
$b \to 1$	no, finite	no
$a \to 0$	logarithmic	yes
$a \to 1$	critical	yes
$y \to 0$	critical	yes
$y \to 1$	no, finite	no
$t \to 0$	square root	yes
$t \to 1$	critical	yes
$c_l \to -1$	square root	yes
$c_l \to 1$	square root	yes
$c_k \to -1$	square root ($t_5 \neq 0$)	yes
	critical ($t_5 = 0$)	yes
$c_k \to 1$	square root ($t_5 \neq 0$)	yes
	critical ($t_5 = 0$)	yes

Table A.1: In this table we lay out the endpoint suppression as implemented in the correlated emission case. The mechanism for the suppression of logarithmic and root divergences was explained in the preceding paragraphs, the suppression mechanism for critical limits will be explained in the next section.

So what about zeroes/divergences which are *not* suppressed, because they are e.g. inside the bulk on the integration region? In these cases the measurement function can be set to a harmless value based on the reasoning that for a logarithmic divergence at x_0 the contribution of the interval $[x_0 - \delta, x_0 + \delta]$ to the full integral vanishes as $\delta \to 0$. We can therefore argue that even in these cases, setting the measurement function to a non-zero value at the critical point should not change the integral estimate, as it is determined by the entire peak, and not just the immediate neighbourhood of the divergence. In practice, the mere existence of the peak changes the way the numerical integrator works, and yields results that are less precise; The integrator essentially tries to sample a divergence using finitely many points. This leads to a reduced numerical accuracy and instabilities if the desired accuracy is set too ambitiously.

A.3 Critical limits

One reason we suppressed the endpoints was so that the integrand function, including all numerator, Jacobian and other functions, tends to zero at the integration boundaries, to avoid potential boundary divergences from contributing and screwing up the numerical integration via infinite variance or other divergence effects.

Unfortunately, the presence of plus-distributions complicates things, so let's recap what a plus-distribution does:

In our calculation the plus distributions appear (here for a fictitious variable x) in the combination

$$[x^{-1+n\epsilon}]_+ R(x) \approx \frac{R(x) - R(0)}{x} + n\epsilon \frac{\ln x}{x}[R(x) - R(0)] + \mathcal{O}(\epsilon^2), \tag{A.4}$$

where $R(x)$ contains, in our case, numerator and measurement functions, Jacobians, etc., and is in general ϵ-dependent, which we shall ignore in this discussion.

Near $x = 0$, we can see the problem if we expand

$$\begin{aligned}[x^{-1+n\epsilon}]_+ R(x) &\approx \frac{R(0) + xR'(0) + \mathcal{O}(x^2) - R(0)}{x} + n\epsilon \frac{\ln x}{x}[R(0) + xR'(0) + \mathcal{O}(x^2) - R(0)] + \mathcal{O}(\epsilon^2) \\ &= R'(0) + n\epsilon \ln(x) R'(0) + \mathcal{O}(x, \epsilon^2).\end{aligned} \tag{A.5}$$

We see that the plus-distribution gets rid of the x^{-1} divergence by throwing away the constant term in the function that multiplies it, enabling a cancellation. The integrand function then has a surviving logarithmic divergence at $x = 0$, which is integrable and a nuisance, from our point of view. Worse, if $R(x)$ doesn't have such a nice expansion, but expands as e.g. $R(x) \approx r_0 + r_1\sqrt{x} + \mathcal{O}(x)$, we'd find

$$\begin{aligned}[x^{-1+n\epsilon}]_+ R(x) &\approx \frac{r_0 + r_1\sqrt{x} + \mathcal{O}(x) - r_0}{x} + n\epsilon \frac{\ln x}{x}[r_0 + r_1\sqrt{x} + \mathcal{O}(x) - r_0] + \mathcal{O}(\epsilon^2) \\ &= \frac{r_1}{\sqrt{x}} + n\epsilon r_1 \frac{\ln x}{\sqrt{x}} + \mathcal{O}(x^0, \epsilon^2),\end{aligned} \tag{A.6}$$

which has a square root divergence, our nemesis when it comes to numerical integration.

Fortunately, the substitutions we used to suppress the integrand at the endpoints also work here. If the function R expands as $R(x) \approx r_0 + r_1 x^m + o(x^m)$, we can substitute $x = y^{\frac{2}{m}}$, and find

$$\begin{aligned}\int_0^1 \mathrm{d}x\, [x^{-1+n\epsilon}]_+ R(x) &= \int_0^1 \mathrm{d}y^{\frac{2}{m}}\, [y^{-\frac{2}{m}+n\frac{2}{m}\epsilon}]_+ R(y^{\frac{2}{m}}) = \frac{2}{m}\int_0^1 \mathrm{d}y\, [y^{-1+n\frac{2}{m}\epsilon}]_+ R(y^{\frac{2}{m}}) \\ &= \frac{2}{m}\int_0^1 \mathrm{d}y\, \frac{R(y^{\frac{2}{m}}) - R(0)}{y} + \frac{2n\epsilon}{m}\frac{\ln y}{y}[R(y^{\frac{2}{m}}) - R(0)] \\ &= \frac{2}{m}\int_0^1 \mathrm{d}y\, \frac{r_0 + r_1(y^{\frac{2}{m}})^m + o((y^{\frac{2}{m}})^m) - r_0}{y} + \frac{2n\epsilon}{m}\frac{\ln y}{y}[R(y^{\frac{2}{m}}) - R(0)] \\ &= \frac{2}{m}\int_0^1 \mathrm{d}y\, r_1 y + o(y) + \frac{2n\epsilon}{m}\frac{\ln y}{y}[r_1 y^2 + o(y^2)] + \mathcal{O}(\epsilon^2) \\ &= \frac{2}{m}\int_0^1 \mathrm{d}y\ r_1\, y + \frac{2n\epsilon}{m}\, r_1\, y \ln y + o(y, \epsilon),\end{aligned} \tag{A.7}$$

which is suppressed at $y = 0$.

A careful analysis of the integrand functions shows that the leading variable dependence in the function R is set — in the actual real world case of our integrand function — by the measurement functions F_X, and that we therefore need to make the parameter m observable dependent. Infrared and collinear safety allow us to determine how the observable scales in most of the critical variables, except for one: The rapidity type variables, i.e. y in the correlated emission case.

We therefore extract m from the functions F_X, unless this is not possible or yields non-sensical results, in which case we use $m = 1$.

Appendix B

Computing parametrisation

In this chapter we list — for completeness' sake — the master formulae in the actual parametrisation used by the program. This form arises after all relevant endpoints are suppressed, as detailed in the previous chapter (following a reparametrisation getting rid of an overlapping divergence in the correlated emission case).

B.1 Correlated emission case

In the correlated emission case we first reparametrise to get rid of the overlapping collinear divergence at $(a, t) = (1, 0)$. To do this we introduce new variables u and v via

$$a = 1 - u(1 - v) \qquad t = \frac{u^2 v}{1 - u(1 - v)}. \tag{B.1}$$

This leads to a critical divergence at $u = 0$, the new form of the collinear divergence, and square root divergences at $u = 1$ and $v = 0$. Having solved the overlapping divergence we reparametrise to suppress the square root and logarithmic divergences for the relevant variables listed in the previous chapter. We therefore introduce:

$$\begin{aligned} u &= 1 - (1 - z^2)^4 & v &= w^4 \\ b &= c^2 & t_l &= 1 - (1 - s_l^4)^4 \\ y &= x^\mu & t_5 &= s_5^2 \end{aligned} \tag{B.2}$$

The exponent μ is related to the parameter m and adjusted to suppress the relevant critical limit. It is set to $\mu = \frac{2}{m}$ if $m \in]0, 1]$, and $\mu = 2$ otherwise.

Using these reparametrisations, and splitting the $C_F C_A$ contribution into a part that matches $C_F T_F n_f$'s divergence structure — called Pseudo-n_f or PNF — and the rest — called Rest-C_A or RCA — we find the master formulae for `SoftSERVE` below. For `SoftSERVE` the monomials marked in blue contribute regulator poles, which are made explicit by introducing plus-distributions. Following this subtraction the complete expressions are expanded in the regulators and can be used for numerical integration. For brevity the argument dependence of the measurement functions F_X has been suppressed, and the labels

$i = 1, 2$ on F_{Xi} correspond to the two distinct substitutions for the angular variable t_k in terms of the angular variables t, t_l and t_5 introduced in [1] (or their resubstituted versions z, w, s_l and s_5 as introduced above).

$$
\begin{aligned}
S_{n_f} = \int_0^1 \mathrm{d}z\,\mathrm{d}c\,\mathrm{d}w\,\mathrm{d}x\,\mathrm{d}s_5\,\mathrm{d}s_l\, & \frac{2^{17-4\epsilon} m\epsilon\Gamma[-2\alpha-4\epsilon]}{e^{2\epsilon\gamma_e}\pi^{\frac{3}{2}}\Gamma[\frac{1}{2}-\epsilon]\Gamma[1-\epsilon]} z^{-1-4\epsilon}\, x^{-1+m\alpha+2nm\epsilon}\, s_5^{-1-2\epsilon} \\
& \cdot \frac{1}{(1+w^4)^3} c^{1-2\alpha-4\epsilon} w^{1-4\epsilon} s_l^{1-4\epsilon} \left(F_{A1}^{4\epsilon+2\alpha} + F_{A2}^{4\epsilon+2\alpha} + F_{B1}^{4\epsilon+2\alpha} + F_{B2}^{4\epsilon+2\alpha}\right) (2-s_5^2)^{-1-\epsilon} \\
& \cdot (1-z^2)^{1-4\epsilon}(1-s_l^4)^{1-4\epsilon} \left((2-s_l^4)(2-2s_l^4+s_l^8)\right)^{-\frac{1}{2}-\epsilon} \left((2-z^2)(2-2z^2+z^4)\right)^{-1-2\epsilon} \\
& \cdot \left(1+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{-\frac{1}{2}-\epsilon} \left((1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{1-\alpha} \\
& \cdot \left(1+c^2\left((1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)\right)^{-2+2\alpha+2\epsilon} \\
& \cdot \left(c^2+(1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{-2+2\epsilon} \Bigg[-4w^4 \Bigg(c^2 + w^4z^2\Big(2-z^2\Big) \\
& \cdot \Big(z^4-2z^2+2\Big) + \Big(1-z^2\Big)^4 \Bigg) \Big(1 + c^2\Big(w^4z^2\big(2-z^2\big)\big(z^4-2z^2+2\big) + \big(1-z^2\big)^4\Big)\Big) \\
& - \big(1-c^2\big)^2\big(1-w^4\big)^2\Big(w^4z^2\big(2-z^2\big)\big(z^4-2z^2+2\big) + \big(1-z^2\big)^4\Big)\Bigg]
\end{aligned}
$$

$$
\begin{aligned}
S_{PNF} = \int_0^1 \mathrm{d}z\,\mathrm{d}c\,\mathrm{d}w\,\mathrm{d}x\,\mathrm{d}s_5\,\mathrm{d}s_l\, & \frac{(\epsilon-1)2^{4(4-\epsilon)} m\epsilon\Gamma[-2\alpha-4\epsilon]}{e^{2\epsilon\gamma_e}\pi^{\frac{3}{2}}\Gamma[\frac{1}{2}-\epsilon]\Gamma[1-\epsilon]} z^{-1-4\epsilon}\, x^{-1+m\alpha+2nm\epsilon}\, s_5^{-1-2\epsilon} \\
& \cdot \frac{1}{(1+w^4)^3} c^{3-2\alpha-4\epsilon} w^{1-4\epsilon} s_l^{1-4\epsilon} \left(F_{A1}^{4\epsilon+2\alpha} + F_{A2}^{4\epsilon+2\alpha} + F_{B1}^{4\epsilon+2\alpha} + F_{B2}^{4\epsilon+2\alpha}\right) (2-s_5^2)^{-1-\epsilon} \\
& \cdot (1-z^2)^{1-4\epsilon}(1-s_l^4)^{1-4\epsilon} \left((2-s_l^4)(2-2s_l^4+s_l^8)\right)^{-\frac{1}{2}-\epsilon} \left((2-z^2)(2-2z^2+z^4)\right)^{-1-2\epsilon} \\
& \cdot \left(1+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{-\frac{1}{2}-\epsilon} \left((1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{1-\alpha} \\
& \cdot \left(1+c^2\left((1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)\right)^{-2+2\alpha+2\epsilon} \\
& \cdot \left(c^2+(1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{-2+2\epsilon} \\
& \cdot \left[2 + w^4z^2\left(2-z^2\right)\left(2-2z^2+z^4\right) - z^2\left(2-z^2\right)\left(2-2z^2+z^4\right)\right]^2
\end{aligned}
$$

$$
\begin{aligned}
S_{RCA} = \int_0^1 \mathrm{d}z\,\mathrm{d}c\,\mathrm{d}w\,\mathrm{d}x\,\mathrm{d}s_5\,\mathrm{d}s_l\,
&\frac{2^{4(4-\epsilon)} m\epsilon\Gamma[-2\alpha-4\epsilon]}{e^{2\epsilon\gamma_e}\pi^{\frac{3}{2}}\Gamma[\frac{1}{2}-\epsilon]\Gamma[1-\epsilon]}\, z^{-1-4\epsilon}\, x^{-1+m\alpha+2nm\epsilon}\, s_5^{-1-2\epsilon} \\
&\cdot c^{-1-2\alpha-4\epsilon}\frac{1}{(1+w^4)} w^{1-4\epsilon} s_l^{1-4\epsilon}\left(F_{A1}^{4\epsilon+2\alpha}+F_{A2}^{4\epsilon+2\alpha}+F_{B1}^{4\epsilon+2\alpha}+F_{B2}^{4\epsilon+2\alpha}\right)(2-s_5^2)^{-1-\epsilon} \\
&\cdot (1-z^2)^{1-4\epsilon}(1-s_l^4)^{1-4\epsilon}\left((2-s_l^4)(2-2s_l^4+s_l^8)\right)^{-\frac{1}{2}-\epsilon}\left((2-z^2)(2-2z^2+z^4)\right)^{-1-2\epsilon} \\
&\cdot \left(1+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{-\frac{1}{2}-\epsilon}\left((1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{-\alpha} \\
&\cdot \left(1+c^2\left((1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)\right)^{-1+2\alpha+2\epsilon} \\
&\cdot \left(c^2+(1-z^2)^4+w^4z^2(2-z^2)(2-2z^2+z^4)\right)^{-1+2\epsilon} \\
&\cdot \Bigg[2c^2\left(w^4z^2\left(2-z^2\right)\left(z^4-2z^2+2\right)+\left(1-z^2\right)^4\right) \\
&\quad -\left(1-\frac{2w^4z^4(2-z^2)^2(2-2z^2+z^4)^2}{w^4z^2(2-z^2)(2-2z^2+z^4)+(1-z^2)^4}\right)\Bigg(c^2 \\
&\quad +2\left(1+c^4\right)\left(w^4z^2\left(2-z^2\right)\left(z^4-2z^2+2\right)+\left(1-z^2\right)^4\right) \\
&\quad +c^2\left(w^4z^2\left(2-z^2\right)\left(2-2z^2+z^4\right)+\left(1-z^2\right)^4\right)^2+c^2\Bigg)\Bigg]
\end{aligned}
$$

Appendix C

Multiprecision

C.1 The problem, explained

To understand the necessity for multiprecision variables, it is best to study a concrete example. Take a measurement function which contains a structure like

$$F(x) = \sqrt{\frac{1}{x} + \frac{2\Delta}{\sqrt{x}}} - \sqrt{\frac{1}{x}} \tag{C.1}$$

where x represents any of the variables that come associated with a plus-distribution, and Δ is some x-independent quantity that may well depend on other variables. For positive Δ this function is finite and positive in the region $x \in [0, 1]$, and has a nontrivial limit as $x \to 0$: $F(0) = \Delta$.

The problem now arises because for small but non-zero x, the two square roots in the difference can become arbitrarily large, but their difference will hover around Δ. In other words, the smaller we choose x, the further down in decimal places is the first digit at which the roots' values will differ. In the above example, for $\Delta = 1$, $x = 10^{-10}$, the result is

$$F(10^{-10}) \approx (1.0000000001 - 1) \times 10^{10} \approx 1 \tag{C.2}$$

This is at odds with the way floating point calculations are performed in any computer program, because for a computer there is a direct relation between the number of digits that are stored for a variable, and the memory that is required to store them. C++ `double` type variables store typically $\approx 15 - 16$ digits[1], and are blind to all digits further down the decimal form. In the above example, if $x \approx 10^{-15}$ or smaller, a computer program would calculate the value of the two square roots and not see that they are different, because the difference occurs only at a digit that is rounded away in `double` type variables. The program would therefore conclude that $F(10^{-15}) = 0$, rather than $F(10^{-15}) \approx \Delta$.

The reason this is a large problem for us, rather than just a small rounding error, becomes clear when plus-distributions come into the fray.

[1]The ambiguity of 15 vs. 16 arises because a computer stores binary numbers, not decimal numbers.

If we apply a $\left[\frac{1}{x}\right]_+$ plus-distribution to F, we find a structure like

$$\left[\frac{1}{x}\right]_+ \ln F(x) = \frac{\ln \frac{F(x)}{F(0)}}{x} \tag{C.3}$$

in the integrand, because F appears only as $F^{4\epsilon}$, which expands to logarithms. $F(0)$ is a non-trivial limit. If we follow the `SoftSERVE` procedures we will have explicitly included an `if`-clause in the input files that makes sure that $F(0)$ is evaluated correctly. But $F(x \neq 0)$ is evaluated using the internal C++ floating point math libraries, and can therefore be subject to the rounding errors we showed above for F.

This is now a two-faceted problem. First, for an F as demonstrated above the program sees that the integrand can contain the schematic expression

$$\left[\frac{1}{x}\right]_+ \ln F(x)\Big|_{x=10^{-15}} = 10^{15} \cdot \ln \frac{F(10^{-15})}{F(0)} = 10^{15} \cdot \ln \frac{0}{\Delta} \to \texttt{NaN}, \tag{C.4}$$

where the logarithm cannot be evaluated, because the program suffered from rounding problems in $F(10^{-15})$. If this occurs, the numerical integrator would return `NaN` as the result of the integration.

But second, and more dangerous, is if we have a measurement function $F'(x) = F(x)+c$, for some constant c, then a computer would return a *wrong* (rather than a *nonsensical*) result, because the computer would calculate

$$\left[\frac{1}{x}\right]_+ \ln F'(x)\Big|_{10^{-15}} = 10^{15} \cdot \ln \frac{0+c}{\Delta+c} = 10^{15} \ln \frac{c}{c+\Delta}, \tag{C.5}$$

rather than the correct result of

$$\left[\frac{1}{x}\right]_+ \ln F'(x)\Big|_{10^{-15}} = 10^{15} \cdot \ln \frac{\Delta+\delta+c}{\Delta+c} \approx -10^{15} \ln 1 \approx 0, \tag{C.6}$$

where δ stands in for the small difference between $F(0)$ and $F(10^{-15})$.

This problem can occur for any measurement function that relies on large cancellations between numbers, and it is not just a theoretical problem; Transverse Thrust is an observable that shows exactly this behaviour.

C.2 The diagnosis

So what can we do to solve the problem?

First we need to have a method of diagnosing the critical feature, to see if a given observable potentially suffers from it. For this we recall that *Divonne* delineates an exclusion zone at the integration boundaries via the `border` setting, in which it does not evaluate the integrand directly. This helps us, because it means that below a certain threshold no variable values will be plugged into any functions. Still, we do not know at which point the rounding errors appear, as this is observable dependent. For some observables $y \sim 10^{-5}$ may trigger them already, for others they may only appear below $y \sim 10^{-20}$. Also recall that the rounding errors do not just cause the integrand to exhibit `NaN` occurrences as the rounding artefacts, but they can also just change its value. Both these artefacts will upset the continuity of the integrand function, and that means that changing the `border` variable will cause a change in the

numerical integration result, because the extrapolation into the boundary region relies on the continuity of the integrand function.

To check for the presence of rounding errors, we therefore perform the calculation with different values for the `border` variable, and if the final result changes we very likely are sensitive to rounding errors, and have to use the multiprecision solution outlined below.

C.3 The solution

The solution to rounding errors due to not resolving enough digits is to resolve more digits.

We can do this by using non-standard datatypes which store more than the C++ standard `double` type's 15 digits. To keep things simple we define a new datatype `decimal`, which is typedef'd to `double` if the usual precision suffices, but which is internally set to a larger datatype, if needed.

To provide the backbone for multiprecision variables, we implement three options:

- The `boost` library[3] provides the `cpp_dec_float_100` datatype, which is implemented via header-only libraries.
- The `GMP` library[4] library provides the `mpf_float_100` datatype, which requires the GMP library to be compiled before use.
- The `MPFR` library[5] library provides the `static_mpfr_float_100` datatype, which requires the GMP and MPFR libraries to be compiled before use.

All of these provide 100 decimal digit precision, and are ordered roughly in performance: `boost` is slowest but easiest to implement, while `GMP` and `MPFR` are fastest but typically increase memory use (`MPFR` in particular).

Because `boost` does not need to be compiled, we use it as the default — how to change the multiprecision backbone to `GMP` or `MPFR` is explained in section 4.10.

Depending on which backbone is chosen, the `decimal` datatype is typedef'd as an alias of the relevant one of the three choices itemised above.

Appendix D

Program structure

Having explained the methods and strategies used for the *calculation*, this chapter serves as the documentation of the *implementation*, i.e. the layout of the code.

This is to a large part dictated by the structure of the calculation itself. The most important of the features of the calculation is that there is a correlation between the dimensionality of the numerical integration and the regulator orders for the first few orders. In detail:

Any master equation in [1] or appendix B (taking SCET-1 without loss of generalisation) can be split schematically into three structures, viz.

$$S = \underbrace{P(\epsilon)}_{\substack{\text{constant}\\ \text{prefactors}}} \left(\Pi_i \int \mathrm{d}x_i\right) \underbrace{X(\{x_i\};\epsilon)}_{\substack{\text{finite}\\ \text{functions}}} \underbrace{\phi\left(F(\{x_i\}), x_i;\epsilon\right)}_{\substack{\text{measurement function}\\ \text{and divergences}}} \tag{D.1}$$

We have split off constant, i.e. integration variable independent, prefactors like π^ϵ or $\Gamma(1-\epsilon)$ from the function to be integrated, and we've collected non-divergent, but regulator dependent functions into the structure X. These mainly are relics of the matrix element numerators, Jacobians and the likes. Finally, ϕ contains the measurement functions F , as well as all divergent monomials (like $y^{-1+n\epsilon}$) before the subtraction.

Two features are now important:

- First, only ϕ contains regulator poles: P and X are finite as $\epsilon \to 0$ (SCET-1) or $\alpha \to 0$, $\epsilon \to 0$ (SCET-2). In the SCET-2 case there may be terms like $\frac{\alpha}{\epsilon}$, but these only exchange the type of the regulator poles. They do not add a net divergence.
- The poles in ϕ only arise from the δ-part in the subtraction $x^{-1+n\epsilon} = \frac{\delta(x)}{n\epsilon} + \left[\frac{1}{x}\right]_+ + n\epsilon \left[\frac{\ln x}{x}\right]_+ + \dots$

We can draw several lessons from this.

D.1 Prefactors

First, note that a given regulator order of the integrand structure $\int X\phi$ contributes to multiple regulator orders of the final result for S, because the prefactor can add any positive power of the regulators. As an example, the lowest regulator order of $\int X\phi$ in the C_A colour structure for an SCET-1 observable has regulator power ϵ^{-4}. This expression, multiplied with the lowest (ϵ^0) order of the prefactor P, appears in the ϵ^{-4} order of the final result soft function. It also appears in the ϵ^{-3} coefficient of the final result when multiplied with the ϵ^1 order of the prefactor, in the ϵ^{-2} coefficient when multiplied with the ϵ^2 order of the prefactor, and so forth.

In other words, a naive expansion in regulator powers for the full soft function master formula (including prefactors) yields expressions for the higher order terms that contain all the lower order expressions, just with different constants multiplying individual functions.

To reduce the number of individual function evaluations the computer has to perform, we therefore split the prefactors from the variable dependent functions, expand the latter and numerically integrate them alone, and then plug the different regulator orders of the numerical results back together with the correct prefactor orders to get the final result coefficients.

D.2 Poles and Deltas

The second point of order concerns the origin of the poles. As each pole reliably comes with a Dirac delta, this means that leading poles arise from integrations with reduced dimensionality of the integration.

Consider as an example again the ϵ^{-4} order of the C_A colour structure for a SCET-1 observable, which is the leading order. The four inverse regulator powers arise in the form $\Pi_i \left(\frac{\delta(x_i)}{\eta_i \epsilon}\right)$ from the subtraction, where the x_i are the relevant integration variables and η_i are constants that differ between the x_i.

To get the result, we therefore do not need to perform the full six-dimensional integral, it suffices to set the four variables that come with a delta to zero analytically, and only perform the surviving two-dimensional integral numerically.

At higher orders this induces a hierarchy. Take the first subleading order of the same colour structure. To get to the ϵ^{-3} pole we can either start with all four deltas from the subtraction, and cancel one of them with an ϵ^1 factor from X or the measurement function F (which appears as $F^{4\epsilon}$ in ϕ), or we can take only three deltas and use the lowest order (ϵ^0) for all other remaining factors from X and F.

In the former case, we only need to perform a 2-dimensional integration, in the latter case we need to perform only 3-dimensional integrations, albeit with terms depending on different sets of three integration variables.[1]

We therefore conclude that for the leading regulator orders the number of integrations depends on the regulator power. In particular, we find that the number of integration dimensions is given by table D.1 The $C_F T_f n_f$ structure only has three divergence monomials, which account for the difference.

One last piece of fortune reduces the number of integration dimensions further. Note that the first finite order (ϵ^0) could arise from the terms in which no delta is present, and all other factors contribute at

[1] There are four different sets of three integration variables, as there are four possibilities of choosing one out of four subtractions. For that one subtraction we *don't* use the delta, but the plus-distribution.

$C_F C_A$ and C_F^2	
regulator powers	integration dimension
-4 (leading)	2
-3	3
-2	4
-1	5
0	6

$C_F T_f n_f$	
regulator powers	integration dimension
-3 (leading)	3
-2	4
-1	5
0	6

Table D.1: *Apparent* number of integration dimensions for $C_F C_A$ and C_F^2 colour structures (left) and $C_F T_f n_f$ structure (right).

$C_F C_A$ and C_F^2	
regulator powers	integration dimension
-4 (leading)	2
-3	3
-2	4
-1	5
0	5

$C_F T_f n_f$	
regulator powers	integration dimension
-3 (leading)	3
-2	4
-1	5
0	5

Table D.2: *Actual* number of integration dimensions for $C_F C_A$ and C_F^2 colour structures (left) and $C_F T_f n_f$ structure (right).

lowest order. In this case it turns out that the relevant expressions from X are independent of one of the integration variables — y in the correlated emission case, b for uncorrelated — which only appears in the measurement function. However, the measurement function appears only as $F^{4\epsilon}$, whose leading order is 1, which obviously is also independent of y and b. We therefore find that the 6-dimensional integral evaluates to zero, because it involves the integral over a plus-distribution $\left[\frac{1}{y}\right]_+$, multiplied with only y-independent functions (b in place of y for uncorrelated emissions).

The number of dimensions of the integrals *actually* required is therefore listed in table D.2.

The case is virtually identical the SCET-2 case, with one minor change:

For SCET-2 observables the appearance of the second regulator, and in particular of terms like $\frac{\alpha}{\epsilon}$, may cause trouble. In practice, careful analysis shows that apart from inflating the number of appearing terms in various expressions, this changes nothing. The dimensionality of the integration is set by the sum of the powers of the two regulators.

D.3 Source files

With what we've learned in the preceding section, and as a primer for the next, we can now list the source files and why they appear where and how they do.

First, note that most source files have an associated header file. The only exceptions are `Master.cpp`, and the input source files. The `Master.cpp` file only contains the `main{}` function, the input files are

headed by `Auxiliaries.h` in the `Sources` subfolder.[2]

All branches of the program have a certain set of source files, namely

`Master.cpp` contains the `main()` function. It is found in the `Sources` subfolder, and starts sequential child processes that perform the numerical integration, one for each integration domain dimensionality. Once an integrator run has finished, these results are combined with the constant prefactors to calculate the soft function Laurent series coefficients.

`Prefactors.cpp` contains the prefactors, which multiply the results of the numerical integration in the Laurent coefficients of the final result soft function. It is also hidden out of view, in the `Sources` folder.

`XFactor.cpp` resides in `Sources` as well, and contains the functions we grouped as X in the discussion above, in their expansion in regulator orders as required by the numerical integration. These files act as libraries for the next set of files.

"`Int`" files in the `Sources` folder adhere to a naming scheme. As an example, the `3IntAM1.cpp` file contains the $\epsilon^i \alpha^{-1}$ orders of the functions that appear in the 3-dimensional Integration, for all relevant ϵ orders i. The α orders are specified by `AM1` and `AM2`, standing in for *Alpha to the Minus 1* or *Alpha to the Minus 2*. `AM0` is omitted wherever it appears.

With the exception of the `2Int` files (or `3Int` for the $T_f n_f$ structure), all of these files contain multiple functions at a given regulator order, corresponding to the different possible subsets of integration variables that can be formed at a given number of integration dimensions. So the `3IntCA` files contain four functions for each regulator order, corresponding to the four possible sets of integration variables (t_l, v, b), (t_l, v, y), (t_l, v, t_5), (t_l, v, u) that span the integration domain, for example.

The Real-Virtual (`RV`) and 1-loop (`OL`) computations only have 2-dimensional integrals, the prefix number is therefore omitted in these cases.

`Auxiliaries.cpp` contains auxiliary function definitions, variable transformations, and the structures that collect the various functions that need to be integrated. The latter takes the different functions defined in the `Int` files and homogenises their integration domain.

"`Input`" files contain all definitions that are observable specific, and integrator parameters. We've already encountered those in the bulk of the manual before.

The `Sources/SCETX` folders contain a subfolder `1AN`, short for "1-particle, C_A and n_f", which contains the files above. There are also currently irrelevant subfolders named `AD`, `S-A`, and `S-B`, which contain no droids anybody might be looking for.

The `1AN` folder includes (almost) multiple sets of these files, marked with `CA` (one part of the $C_F C_A$ structure), `PNF` ("Pseudo-n_f", second part of the $C_F C_A$ structure, it behaves like the $C_F T_f n_f$ structure), `NF` ($C_F T_f n_f$ structure), `RV` ("Real-Virtual", i.e. 1-particle NNLO), `OL` ("One Loop", i.e. NLO) and `1P` ("1-particle" in general).

All these files share one subfolder as there are shared source files: the `Input` and `Auxiliaries.cpp` files.

To get more familiar with the content of these files it is instructive to see them in action, so we run along a typical program run, and explain along the way what happens.

[2] With the exception of the content of `Input_Parameters.h`, of course.

D.4 Program flow

The schematic flow is described in figure D.1, where the $C_F C_A$ or C_F^2 structures serve as the template. The other colour structures are similar with minor modifications.

Having defined the relevant input in the `Input` files and compiled the program, we call - in a typical run - the binary for a given colour structure.

The `main` function then spawns a child process[3] for the lowest dimensional integration: For all colour structures except $C_F T_f n_f$ this is the 2-dimensional integration, $C_F T_f n_f$ starts at the 3-dimensional integration

This child process sets up variables for the integration results, and calls the `llDivonne` function as defined in the Cuba library to integrate all regulator orders required for any Soft function Laurent series coefficient in parallel, to fill these variables.

To do this, `Divonne` uses the functions defined in the "`Int`" files, which in turn depend on the measurement functions F or G and parameters n and m defined in the `Input` files and the X functions in `XFactor.cpp`. The functions from the "`Int`" files are funnelled through `Auxiliaries.cpp` where the different integration variable sets are aligned[4]

The integration then proceeds according to the parameters specified in the `Input` files.

Once the first integration is finished, by the arguments explained in the previous chapter, enough information is available to compute the leading pole coefficient from the integration result and the prefactors in `Prefactors.cpp`.

The 1-particle calculation terminates here, as it only needs 2-dimensional integrals. For all other colour structures we proceed to the second integration, which has one integration variable more[5]. There, the same procedure happens. Once this integration finishes, enough information is available for the first subleading pole.

The whole process repeats until the 5-dimensional integration is finished. When this happens the program tidies up, calculates the finite and linear regulator divergence coefficients, and terminates.

The procedures for SCET-2 are identical, except that the appearance of terms is set by the sum of the ϵ and α orders. So e.g. for C_F^2 the $\epsilon^{-4}\alpha^0$, $\epsilon^{-3}\alpha^{-1}$ and $\epsilon^{-2}\alpha^{-2}$ pole coefficients all count as leading and are available once the 2-dimensional integration has terminated.

[3] the reason for using child processes is technical and laid out in the comments in the `Master.cpp` files

[4] If e.g. multiple functions contribute to for example the 3-dimensional integration, which depend on different sets of three integration variables, e.g. (t_l, v, b), (t_l, v, y), etc., `Auxiliaries.cpp` provides one function depending on variables (x_1, x_2, x_3) which is the sum of all 3-variable functions in the "`Int`" file.

[5] i.e. 4-dim for $C_F T_f n_f$, 3-dim for everybody else.

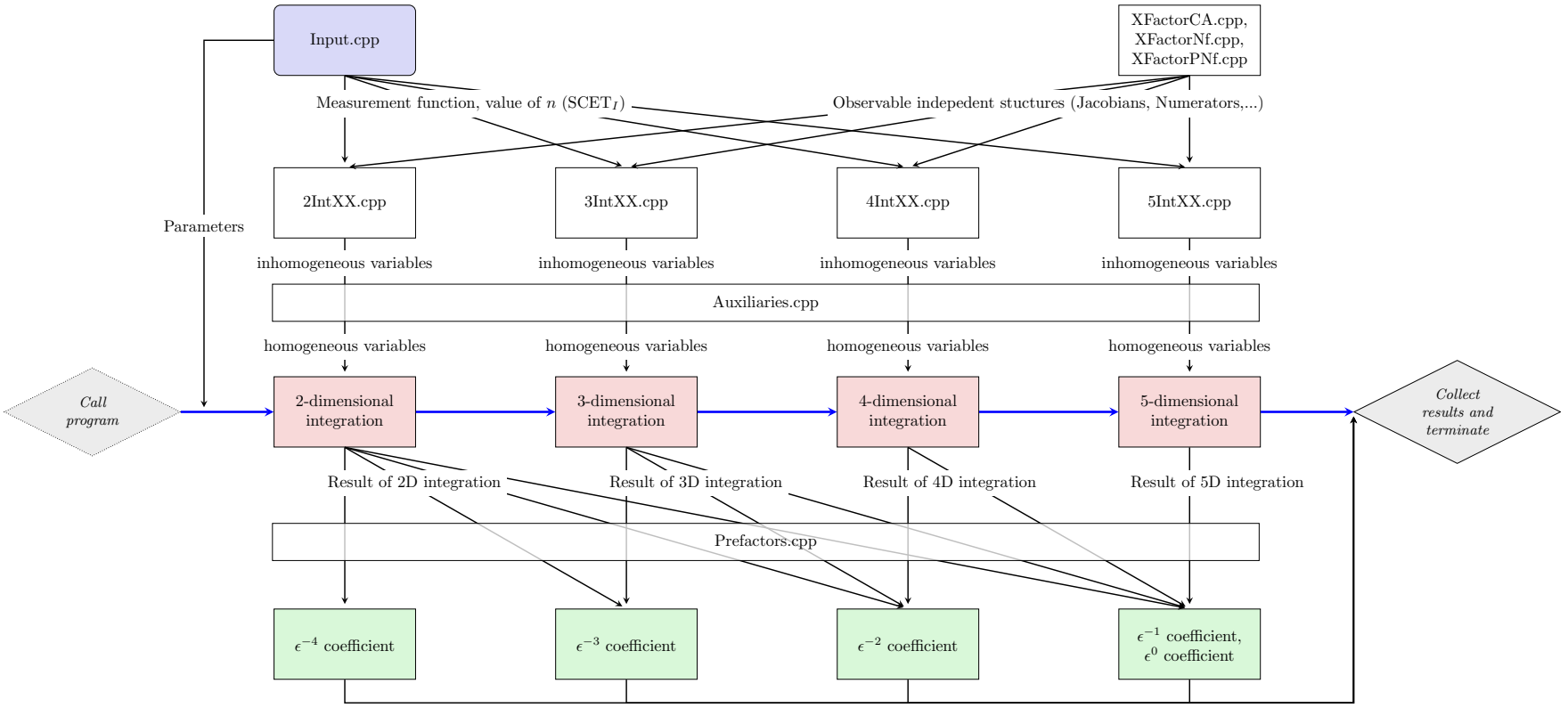

Figure D.1: Program flow through the program for a SCET-1 observable. The blue box marks the only files requiring user input, red boxes represent program steps, white boxes mark the source files. Green boxes represent results. The blue arrows trace the program flow, black arrows of the form $A \to B$ represent "Variables in A passed to B" or "Function defined in A called by B". The chart should be read left-to-right and top-down, and external libraries are not included.

Appendix E

Template Observables

E.1 Preliminaries

Below we provide the derivation of the measurement functions for the various SCET-1 and SCET-2 observables included as pedagogical template examples in the `SoftSERVE` package.

The ultimate aim is to derive the functions F_X, with placeholder index X, to be used in the `SoftSERVE` program. We will use the term "measurement function" interchangeably for the F functions as well as the full measurement exponential, the meaning is in general clear from the context.

To recap the main manual and articles:

- The initial measurement function — we call it F_A— can be derived from the measurement exponential using the appropriate parametrisation outlined in section 4.2, followed by stripping off some terms, mainly leading powers in certain limits.
- The function F_B arises from the measurement exponential the same way as the function indexed A, after inverting one suitable variable, according to the symmetry considerations outlined in [1].
- The F_X functions' behaviour around $y = 0$ determines the parameter m. If that doesn't work, we use $m = 1$.

We run through the examples below, but it is advisable to list briefly which issues in general require attention:

- Non-trivial limits (e.g. of the form $\frac{0}{0}$) must be caught by `if`-clauses
- Limits in which the measurement function vanishes and/or diverges must be explicitly "defused" by setting the measurement function equal to a constant[1] at these points.
- Heaviside step functions are best implemented using `if`-clauses. This has the advantage that any considerations of `NaN` appearances or zeroes/divergences can be confined to the regions where the `if`-clause triggers, i.e. where the step functions has support.

[1] Setting them to 1 is computationally sensible, some terms drop out of the integrand then.

E.2 First appearance of critical features

Many observables share intricate features. To keep this manual short, we generally do not explain in detail how a certain issue is handled, if another observable covered in the sections before also exhibits it. Instead, here is a table of interesting features that lists for which observable we've covered them in detail.

Feature	Example observable	Section
Basic vanilla observable	Threshold Drell-Yan	E.3
Different regions A, B	C-Parameter	E.4
Vanishing/diverging F_X	C-Parameter	E.4
Parameter dependence	Angularities	E.5
Non-trivial variable m	Angularities, $W/Z/H$ at large p_T	E.5 E.7
Step functions	Angularities	E.5
Angular dependence	Gauge boson at large p_T	E.7
Non-standard RG equation	Transverse Thrust	E.8
Non-trivial limit	Transverse Thrust	E.8
Rounding errors	Transverse Thrust	E.8
Imaginary F	p_T resummation	E.9

Table E.1: Observables illustrating the first appearances of non-trivial `SoftSERVE` features.

Warning!

The symmetry discussion in the main paper shows explicitly that there are four distinct integration regions covered by the two regions A and B, each. The symmetry enforces that these give the same result *upon integration*, allowing us to get away with only calculating the results in A and B, and accounting for the others by multiplying these results by 4.

The symmetry does ***not*** enforce equality of the *integrands* in these mirror regions. Depending on how you derive F_B from F_A or the measurement exponential, you may get different functional forms for F_B. Symmetry then states that these must give the same final result.

So do not be alarmed if there are different F_B functions you can derive. Chances are they yield the same result.

E.3 Threshold Drell-Yan

The soft function for Threshold Drell-Yan[7] involves the measurement exponential

$$\exp(-\tau\omega^{(2)}(k,l)) = \exp(-\tau(k_+ + k_- + l_+ + l_-)), \tag{E.1}$$

as derived in the initial sections of the main paper.

We can now apply the parametrisations, and read off the measurement function, as well as - in the correlated emission case - the parameter n:

$$\exp(-\tau\omega^{(2)}(k,l)) = \exp(-\tau\, p_T\, \frac{1+y}{\sqrt{y}}) \tag{E.2}$$

To derive F_B we need to invert one of a, b, y, which has no effect.

We conclude that for the correlated emission:

$$n = -1 \qquad F_A = F_B = 1 + y \tag{E.3}$$

The leading non-constant power in y can be easily read off: $F_{A/B} = const. + \mathcal{O}(y)$, so following the manual we have $m = 1$.

Problematic limits

There are none, this function is well behaved in the entire integration space.

E.4 C-Parameter

The C-Parameter soft function[8] is similar to the Threshold Drell-Yan case:

$$\mathcal{S}(\omega) = \sum_{k,l} \langle 0|\, S^\dagger S\, |k,l\rangle\, \langle k,l|\, SS^\dagger\, |0\rangle\ \delta(\omega - \frac{k_+k_-}{k_+ + k_-} - \frac{l_+l_-}{l_+ + l_-}). \tag{E.4}$$

Following a Laplace transform we find for the correlated emission case the measurement exponential

$$\mathcal{M}_{CP,A} = \exp(-\tau\, p_T\, \sqrt{y}(\frac{a}{a+b+y(1+ab)} + \frac{ab}{a(a+b)+y(1+ab)})). \tag{E.5}$$

From this we can read off

$$n = 1 \qquad F_A = \frac{a}{a+b+y(1+ab)} + \frac{ab}{a(a+b)+y(1+ab)} \tag{E.6}$$

The function in region B is different, now, and to get it we can either invert y, a, or b[2]. This yields

$$\mathcal{M}_{CP,B} = \exp(-\tau\, p_T\, \sqrt{y}(\frac{a}{a(1+ab)+y(a+b)} + \frac{ab}{1+ab+ay(a+b)})). \tag{E.7}$$

and so

$$F_B = \frac{a}{a(1+ab)+y(a+b)} + \frac{ab}{1+ab+ay(a+b)} \tag{E.8}$$

Expanding F_X around $y = 0$ yields the leading non-constant power in y, again we have $F_X = const. + \mathcal{O}(y)$, so again we find $m = 1$.

[2]Either will do, choose the one that causes you the least amount of work.

Problematic limits

The measurement functions in the correlated emission case vanish if $a = 0$. This is a lower dimensional hypersurface and not a limit associated with a divergence/plus-distribution, and therefore acceptable. Nevertheless the program will throw up error messages if this is not handled. We therefore include a segment

```
if(a<=0.) {
   FA=1.;
}
```

for F_A and similar for F_B in their sections of the `Input_Measurement_Correlated.cpp` file.

For the reasons why changing the value of the function is acceptable, please consult appendix A.

Additionally, there are overlapping non-trivial limits of the form $\frac{0}{0}$, e.g. in F_A's first term, at $(a, b, y) \to (0, 0, 0)$. If C++ tried to evaluate the function there, it would encounter the expression $\frac{0}{0+0+0} + \ldots$, which it can't resolve on its own. However, all of these require $a \to 0$, and therefore are already covered by the if clause above.

This exhausts the list. The functions are well-behaved everywhere else.

E.5 Angularities

Angularities[9] is again of a form requiring a Laplace transform, yielding

$$\mathcal{M}_{Ang} = \exp(-\tau\, \omega(k, l)), \tag{E.9}$$

where $\omega(k, l)$ is the definition of the angularities observable in terms of emission momenta.

This definition reads:

$$\omega(k, l) = \Theta(k_+ - k_-)\, k_-^{1-\frac{A}{2}}\, k_+^{\frac{A}{2}} + \Theta(k_- - k_+)\, k_+^{1-\frac{A}{2}}\, k_-^{\frac{A}{2}} \tag{E.10}$$

$$+ \Theta(l_+ - l_-)\, l_-^{1-\frac{A}{2}}\, l_+^{\frac{A}{2}} \quad + \Theta(l_- - l_+)\, l_+^{1-\frac{A}{2}}\, l_-^{\frac{A}{2}}. \tag{E.11}$$

As we can see, the angularities are parameter dependent, which must be properly declared to the program, and assigned a value, as `SoftSERVE` cannot handle parametric expressions. We therefore have to add a line to `Input_Common.cpp` and `Input_Parameters.h` each:

```
extern double AA;
```

in `Input_Parameters.h` to declare the parameter and

```
double AA=0.5;
```

in `Input_Common.cpp` to define it. We chose `AA` instead of `A` because the letter A is already in use, it is used by one of the internal functions. For a list of parameter names to avoid, see Table 4.1.

Plugging in the parametrisation for correlated emission, we find for region A

$$\omega_A(k,l) = p_T y^{\frac{1-A}{2}} \Bigg(\Theta(\frac{a(a+b)}{1+ab} - y) \quad \Big[a^{-\frac{A}{2}} (a+b)^{-1+\frac{A}{2}} (1+ab)^{-\frac{A}{2}} (a+a^A b) \Big] + \Theta(y - \frac{a(a+b)}{1+ab}) \Big[a^{1-\frac{A}{2}} \Big((a+b)^{-1+\frac{A}{2}} (1+ab)^{-\frac{A}{2}} + b(a+b)^{-\frac{A}{2}} (1+ab)^{-1+\frac{A}{2}} y^{-1+A} \Big) \Big] \Bigg),$$

where we already used that some of the step functions will always evaluate to 0 or 1 if the variables a, b and y are in $[0,1]$. We also use that $\Theta(-x) = 1 - \Theta(x)$.

For region B we again invert one of y, a, or b in the full exponential, and find

$$\omega_B(k,l) = p_T y^{\frac{1-A}{2}} \Bigg(\Theta\left(\frac{a(1+ab)}{a+b} - y\right) \quad \Big[a^{-\frac{A}{2}} (a+b)^{-\frac{A}{2}} (1+ab)^{-1+\frac{A}{2}} (a^A + ab) \Big] + \Theta\left(y - \frac{a(1+ab)}{a+b}\right) \Big[a^{1-\frac{A}{2}} \Big(b(a+b)^{-\frac{A}{2}} (1+ab)^{-1+\frac{A}{2}} + (a+b)^{-1+\frac{A}{2}} (1+ab)^{-\frac{A}{2}} y^{-1+A} \Big) \Big] \Bigg),$$

where now different step functions evaluate to always 0 or 1.

From this we can read off that $n = 1 - A$, and we can easily identify the functions F_A and F_B . We also brought the entire expression in a form where only ever one of the step functions is non-vanishing, which matches an `if-else` combination.

Our `Input_Measurement_Correlated.cpp` file will therefore contain the code fragment

```
if (y <= a*(a+b)/(1+a*b) ) {
   FA=pow(a+b,-1+AA/2.)*(a+pow(a,AA)*b)*(pow(a,-AA/2.)
      *pow(1+a*b,AA/2.));
} else {
   FA=pow(a,1-AA/2.)*(pow(a+b,-1+AA/2.)*pow(1+a*b,-AA/2.)
      +b*pow(1+a*b,-1+AA/2.)*pow(y,-1+AA)*pow(a+b,-AA/2.));
}
```

for F_A, and similar for F_B.

The behaviour for small y is flat, thanks to the step functions, except if $a = 0$, which is suppressed. This means that expanding $F_X - F_X|_{y=0}$ in the region near $y = 0$ doesn't give us a well-defined value for m. We therefore use $m = 1$.

Problematic limits

First we note that $A = 1$ corresponds to a SCET$_{II}$ type observable, the SCET$_{II}$ branch of `SoftSERVE` must be used for that parameter value, accordingly, for all other values we can simply use the SCET$_I$ branch and adjust the parameter for different runs.

There is further again a zero/divergence in the correlated emission functions at $a = 0$, which we catch the same way as for the C-parameter. This renders the functions well-behaved, any other zero/divergence is overlapping and needs $a = 0$, or is killed by the `if`-clause (e.g. $y \to 0$).

E.6 Thrust

Thrust is merely a special case of the angularities with $A = 0$, any and all relevant input can be easily derived.

E.7 Gauge boson production at large p_T

Weak gauge boson production at large transverse momentum is one of the easiest observables exhibiting angular dependence. The origin of this dependence is the presence of a third jet against which the boson recoils. It turns out[10] that although this third jet does not give additional diagrammatic contributions, its presence breaks rotational invariance around the beam axis, which introduces the angular dependence.

As before, the Laplace space soft function has the correct form measurement function for `SoftSERVE`, with[10]

$$\omega(k,l) = n_j \cdot (k+l) = k_+ + k_- - 2\sqrt{k_+ k_-}\cos\theta_k + l_+ + l_- - 2\sqrt{l_+ l_-}\cos\theta_l \tag{E.12}$$

Using the correlated emission parametrisation, we find

$$\omega(k,l) = p_T y^{-\frac{1}{2}}\left(1 + y - 2\sqrt{\frac{ay}{(a+b)(1+ab)}}\,(b\,c_k + c_l)\right). \tag{E.13}$$

Inverting any of y, a or b has no effect up to relabelling c_k and c_l, so

$$n = -1 \qquad F_A = F_B = 1 + y - 2\sqrt{\frac{ay}{(a+b)(1+ab)}}\,(b\,c_k + c_l) \tag{E.14}$$

$$= 1 + y - 2\sqrt{\frac{ay}{(a+b)(1+ab)}}\,(b\,(1-2t_k) + 1 - 2t_l) \tag{E.15}$$

where the angular dependence can be written in terms of both c_i or t_i variables, however you prefer.

Finally, it is readily apparent that $F_X = c_0 + \sqrt{y}\,c_1 + \ldots$, and so $m = 0.5$.

Problematic limits

There's a zero hiding at $y = a = c_l = 1$, $b = 0$ and a potentially difficult limit at $a = b = 0$ in the correlated emission, which can be dealt with by setting the function to a constant if either $a = 0$ or $c_l = 1$. The other variables involved do not need to be considered, the $a = 0$ and $c_l = 1$ limits are suppressed on their own.

E.8 Transverse Thrust, SCET-1

Transverse Thrust[11] is a hadron collider dijet observable, and therefore in principle needs four Wilson lines (2 jets, 2 beams) in the Soft function. It is, however, possible to reconstruct the anomalous dimension from related observables: leptonic dijet ($0 \to 2$ jets) and hadronic 0-jet ($2 \to 0$ jets) Transverse

Thrust. The former is a SCET-1 observable, the latter is of SCET-2 type. Here we consider the SCET-1 component. The SCET-2 component can be found below.

Non-standard RG equation

Transverse Thrust is one of the most computationally complicated observables we've treated so far, with its functional dependence on the emission momenta reading

$$\omega(k,l) = \sum_{p=k,l} \frac{1}{2|s|} \sqrt{\left((p_- - p_+)s + 2c\, c_p \sqrt{p_+ p_-}\right)^2 + 4(1-c_p^2)p_+ p_-} \tag{E.16}$$

$$- \frac{1}{2|s|} \left|(p_- - p_+)s + 2c\, c_p \sqrt{p_+ p_-}\right|, \tag{E.17}$$

where s and c are the sine and cosine of the angle between beam- and jet- axes, and are used as parameters, and must be declared and defined as usual[3].

Naively we would now apply parametrisations, read off parameters and measurement functions, and run through the whole sequence of steps to get the bare soft function. And then we'd encounter a problem:

So far, all of our observables had measurement exponentials of the form

$$\mathcal{M} = \exp(-\tau\omega(k,l)), \tag{E.18}$$

and they fulfilled Laplace space RG equations of the form

$$\frac{dS(\tau)}{\ln\mu} = \frac{1}{n}\left[\gamma_s + \Gamma_{Cusp} \ln \mu\tau e^{\gamma_e}\right] S(\tau). \tag{E.19}$$

For this type of observables we have a script (`./laprenorm`), which does the extraction of the 2-loop contribution to γ_s much quicker than we could ever do manually.

Transverse Thrust in [11], however, fulfils a Laplace space RGE of the form[4]

$$\frac{dS_{TT}(\tau)}{\ln\mu} = \frac{1}{n}\left[\gamma_s + \Gamma_{Cusp} \ln \frac{\mu\tau e^{\gamma_e}}{4s^2}\right] S_{TT}(\tau). \tag{E.20}$$

If we use the scripts provided, this would amount to shifting a term $-\Gamma_{Cusp} \ln 4s^2$ to the non-cusp part of the anomalous dimension. We'd have to manually account for that.

Alternatively we can define $\tilde{\tau} = \frac{\tau}{4s^2}$, and use the fact that the τ that appears in the RGE also appears in the measurement exponential. We can therefore account for multiplicative factors by shifting them between τ and the functions F and G which multiply it.

We therefore perform such a shift, and use the measurement exponential

$$\mathcal{M} = \exp(-\tilde{\tau}\tilde{\omega}(k,l)), \tag{E.21}$$

[3] see E.5

[4] See its eq. 4.5, where κ maps onto our Laplace space $\bar{\tau}^{-1}$.

with

$$\tilde{\omega}(k,l) = \sum_{p=k,l} 2s\sqrt{\left((p_- - p_+)s + 2c\,c_p\sqrt{p_+p_-}\right)^2 + 4(1-c_p^2)p_+p_-} \tag{E.22}$$

$$- 2s\left|(p_- - p_+)s + 2c\,c_p\sqrt{p_+p_-}\right|, \tag{E.23}$$

Using the correlated emission parametrisation we find

$$\begin{aligned}\tilde{\omega}(k,l) = \frac{4s\,p_T}{\sqrt{y}}\Bigg[&b\sqrt{\left(\frac{as}{2(1+ab)} + \frac{c\,c_k\sqrt{ay}}{\sqrt{(a+b)(1+ab)}} - \frac{sy}{2(a+b)}\right)^2 + \frac{ay(1-c_k^2)}{(a+b)(1+ab)}}\\ &- b\left|\frac{as}{2(1+ab)} + \frac{c\,c_k\sqrt{ay}}{\sqrt{(a+b)(1+ab)}} - \frac{sy}{2(a+b)}\right|\\ &+\sqrt{\left(\frac{s}{2(1+ab)} + \frac{c\,c_l\sqrt{ay}}{\sqrt{(a+b)(1+ab)}} - \frac{asy}{2(a+b)}\right)^2 + \frac{ay(1-c_l^2)}{(a+b)(1+ab)}}\\ &- \left|\frac{s}{2(1+ab)} + \frac{c\,c_l\sqrt{ay}}{\sqrt{(a+b)(1+ab)}} - \frac{asy}{2(a+b)}\right|\Bigg].\end{aligned} \tag{E.24}$$

Naively we would now conclude that $n = -1$ due to the leading factor in the string of functions. However, an expansion around $y = 0$ shows that $\omega \sim \sqrt{y}$ to leading order. We must therefore factor out a factor of $\sqrt{y}$, i.e. $n = 1$. Inverting y, a or b (and exchanging c_k, c_l if b is inverted), we find and read off

$$\begin{aligned}F_A = 4s\Bigg[&b\sqrt{\left(\frac{as}{2(1+ab)y} + \frac{c\,c_k\sqrt{a}}{\sqrt{(a+b)(1+ab)y}} - \frac{s}{2(a+b)}\right)^2 + \frac{a(1-c_k^2)}{(a+b)(1+ab)y}}\\ &- b\left|\frac{as}{2(1+ab)y} + \frac{c\,c_k\sqrt{a}}{\sqrt{(a+b)(1+ab)y}} - \frac{s}{2(a+b)}\right|\\ &+\sqrt{\left(\frac{s}{2(1+ab)y} + \frac{c\,c_l\sqrt{a}}{\sqrt{(a+b)(1+ab)y}} - \frac{as}{2(a+b)}\right)^2 + \frac{a(1-c_l^2)}{(a+b)(1+ab)y}}\\ &- \left|\frac{s}{2(1+ab)y} + \frac{c\,c_l\sqrt{a}}{\sqrt{(a+b)(1+ab)y}} - \frac{as}{2(a+b)}\right|\Bigg]\\ F_B = 4s\Bigg[&b\sqrt{\left(\frac{as}{2(1+ab)} + \frac{c\,c_k\sqrt{a}}{\sqrt{(a+b)(1+ab)y}} - \frac{s}{2(a+b)y}\right)^2 + \frac{a(1-c_k^2)}{(a+b)(1+ab)y}}\\ &- b\left|\frac{as}{2(1+ab)} + \frac{c\,c_k\sqrt{a}}{\sqrt{(a+b)(1+ab)y}} - \frac{s}{2(a+b)y}\right|\\ &+\sqrt{\left(\frac{s}{2(1+ab)} + \frac{c\,c_l\sqrt{a}}{\sqrt{(a+b)(1+ab)y}} - \frac{as}{2(a+b)y}\right)^2 + \frac{a(1-c_l^2)}{(a+b)(1+ab)y}}\\ &- \left|\frac{s}{2(1+a)b} + \frac{c\,c_l\sqrt{a}}{\sqrt{(a+b)(1+ab)y}} - \frac{as}{2(a+b)y}\right|\Bigg],\end{aligned} \tag{E.25}$$

When we use these formulae in `Input_Measurement_Correlated.cpp` we now must explicitly insert an `if`-clause to resolve the limit $y \to 0$, because cmath can of course not simply evaluate the function there. The limit of $y \to 0$ is for e.g. F_A:

$$F_A|_{y\to 0} = \frac{4a(1-c_l^2) + 4b(1-c_k^2)}{(a+b)} \tag{E.26}$$

Accordingly the function definition in `Input_Measurement_Correlated.cpp` takes the schematic form (double brackets are short for entire code segments)

```
double FA(decimal v,decimal t6,decimal y,decimal t5,decimal u,decimal b) {
[[variable definitions]]

   if ( y<=0. ) {
      FA= [[analytic limit for y=0]];
   } else {
      FA= [[Full expression]];
   }

   if ( a<=0. || cl>=1. || cl<=1. ) {
      FA=1.;
   }

[[predefined error corrections and return]]
}
```

where we again introduced a clause capturing the zeroes that are present when either $a \to 0$ or $(c_l, c_k) \to (\pm 1, \pm 1)$, or $(c_l, b) \to (\pm 1, 0)$. Only capturing a and c_l is enough, as we know from our thorough reading of the manual's appendices that the integrand vanishes as $a \to 0$, or $c_l \to \pm 1$. We do therefore not have to specify that the problematic points only affect submanifolds of the $c_l = 0$ hyperplane[5]. The entire hyperplane does not contribute. We can skip the second if-clause nailing down the $c_k = \pm 1$ or $b = 0$ constraints, which would in principle be required to fully localise the zero, as well.

To top off the list of difficulties, expanding the functions F around $y = 0$ shows that we have $m = 0.5$, as for section E.7.

Problematic limits

As already mentioned the non-trivial and non-zero limits of $y \to 0$ must be handled properly.

Also there are zeroes at $a = 0$, or $|c_l| = 1$ together with certain values for b and c_k , which must be captured in the code, as done above.

Rounding errors - multiprecision

Finally, the potentially large cancellation between root and absolute value is a problem for computer-based methods, because the usual data types (`double`, mainly) only resolve a certain finite number of digits. For Transverse Thrust this is an actual problem, and the "vanilla" flavour of `SoftSERVE` is not

[5]The submanifolds being the $b = 0$ or $ck = \pm 1$ hyper^{2}planes.

able to calculate this bare soft function. Using more digits solves this problem, we therefore need to call `make [target] BOOST=1` to compile the executable for whatever `[target]` colour structure we need.

The default multiprecision backbone is provided by the header-only *boost* library. Two additional flavours of GMP are implemented via the *boost* library, but these need to be compiled before use, and are therefore not set as the default. More information can be found in the main manual.

E.9 p_T resummation

Apart from being the first SCET$_{II}$ case in this list, the soft function relevant for transverse momentum resummation is schematically not terribly difficult, compared to others, with one exception: as it naturally arises in Fourier space, its measurement function includes an imaginary unit:

$$\mathcal{M}(k,l) = \exp(-i\tau\omega(k,l)) = \exp(-2i\tau p_T \sqrt{\frac{a}{(a+b)(1+ab)}}\,(c_l + bc_k)) \tag{E.27}$$

It therefore falls under the special case outlined in appendix A of the `SoftSERVE` manual, and accordingly we provide the source files for the required input, which is a `SoftSERVE` run on the input involving the measurement function $|F|$, which yields $n = 0$ (i.e. SCET$_{II}$), and

$$F_A = F_B = 2\sqrt{\frac{a}{(a+b)(1+ab)}}\,|c_l + bc_k| = 2\sqrt{\frac{a}{(a+b)(1+ab)}}\,|1 - 2t_l + b(1-2t_k)| \tag{E.28}$$

We merely have to remember that after getting the results for this observable, we have to run the `./fourierconvert` script before renormalising.

Non-trivial limits

As so often, there is a zero at $a = 0$, which we handle as usual. There is also an overlapping zero at $c_l = \pm 0$ with appropriate values for b or c_k. This latter zero resides in the bulk of the integration region, but only constitutes a lower dimensional hypersurface, which has measure zero. We therefore simply set the measurement function to 1 for relevant points. Additionally there is a non-trivial limit at $(a, b) = (0, 0)$, which is already covered by the $a = 0$ zero.

E.10 Other observables

The other template examples we provide do not contain any difficulties we haven't addressed here yet, and all appearing features are found in other observables, as listed in table E.1.

Bibliography

[1] Guido Bell, Rudi Rahn, and Jim Talbert. *Generic dijet soft functions at NNLO: correlated emissions*. in preparation.

[2] Thomas Hahn. "CUBA: A Library for multidimensional numerical integration". In: *Comput. Phys. Commun.* 168 (2005). Korobov numbers set to 16033 numbers file, pp. 78–95. arXiv: hep-ph/0404043 [hep-ph]. URL: http://www.feynarts.de/cuba/.

[3] The boost library developer team. *boost C++ libraries*. 1.66.0. Version 1.66.0, retrieved 6/2/2018 (2/6/2018 for Americans). 2018. URL: http://www.boost.org/.

[4] Torbjörn Granlund and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library*. 6.1.2. 2016. URL: http://gmplib.org/.

[5] Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. "MPFR: A Multiple-precision Binary Floating-point Library with Correct Rounding". In: *ACM Trans. Math. Softw.* 33.2 (June 2007). Version 4.0.0, retrieved 6/2/2018 (2/6/2018 for Americans). URL: http://www.mpfr.org/.

[6] The boost library developer team. *boost library documentation, Performance comparison*. 2017. URL: http://www.boost.org/doc/libs/1_66_0/libs/multiprecision/doc/html/boost_multiprecision/perf/realworld.html.

[7] Andrei V. Belitsky. "Two loop renormalization of Wilson loop for Drell-Yan production". In: *Phys. Lett.* B442 (1998), pp. 307–314. arXiv: hep-ph/9808389 [hep-ph].

[8] André H. Hoang, Daniel W. Kolodrubetz, Vicent Mateu, and Iain W. Stewart. "C-parameter distribution at N^{3}LL' including power corrections". In: *Phys. Rev.* D91.9 (2015), p. 094017. arXiv: 1411.6633 [hep-ph].

[9] Andrew Hornig, Christopher Lee, and Grigory Ovanesyan. "Effective Predictions of Event Shapes: Factorized, Resummed, and Gapped Angularity Distributions". In: *JHEP* 05 (2009), p. 122. arXiv: 0901.3780 [hep-ph].

[10] Thomas Becher, Guido Bell, and Stefanie Marti. "NNLO soft function for electroweak boson production at large transverse momentum". In: *JHEP* 04 (2012), p. 034. arXiv: 1201.5572 [hep-ph].

[11] Thomas Becher and Xavier Garcia i Tormo. "Factorization and resummation for transverse thrust". In: *JHEP* 06 (2015), p. 071. arXiv: 1502.04136 [hep-ph].