Preprint typeset in JHEP style - PAPER VERSION

The POWHEG BOX V2 framework

The POWHEG BOX V2 authors

E-mail: Paolo.Nason@mib.infn.it

Abstract: This is a very preliminary draft of the V2 paper. It will eventually become a publication. It is made public because many features of the V2 framework are illustrated here.

Keywords: POWHEG, SMC, NLO, QCD.

Contents

1. Overview **2**

2. Treatment of resonances **5**
2.1 Implementation of resonances decays 7
2.2 Internal implementation 8
2.2.1 Finding the singular regions 8
2.2.2 Real phase space 10
2.2.3 Collinear and soft limit of the real cross section 11
2.2.4 Integrated soft corrections 11

3. The fastbtlbound feature **12**

4. The storemintupb feature **13**

5. Parallel runs **16**
5.1 Files to inspect at the end of the run 18
5.2 Shower and analysis 19
5.3 A typical example of a script to perform parallel runs 20

6. Massive emitter **21**
6.1 Implementation in the code 23

7. Photon radiation **23**
7.1 Finding the singular regions 23
7.2 Computation of the NLO cross section 24
7.3 Generation of radiation 25

8. The reweighting feature **25**
8.1 How to use the reweighting feature 26
8.2 Implementation of the reweighting feature 28

A. Basic POWHEG formulae **29**

B. Generation of final state radiation **30**

1. Overview

In this paper we document the new features implemented in the `POWHEG BOX V2`. The `POWHEG BOX V2` is an extension of the `POWHEG BOX` package, featuring many enhancements and improvements. In the following, we will refer to the original `POWHEG BOX` package as `V1`, and to the new version as `V2`. Since when it was first made public, several improvements where added to the `V1`. They were accessible by following detailed instructions, that typically involved replacing some `V1` files. These new additions were not always consistent among each other, i.e. one often could not use more than one such feature at the time. In `V2` all these additions have been merged, and made compatible.

Processes that were developed with `V1` should be easily portable to `V2`. Furthermore, if the new features introduced in `V2` are switched off, the output of the `V2` and `V1` implementations should differ only at the 5 or more digits level, making it easy to test that the port was successful.

The most important improvements in `V2` are the following:

- **Resonance decays.** This is especially relevant if one wants to consider the decay of resonances into hadrons. The `POWHEG BOX` machinery, when seeing quarks or gluons in the final state, considers them as being candidates to form collinear singularities with other (final state or initial state) partons. On the other hand, partons coming from a decaying resonance cannot form collinear pairs with partons not arising from the same resonance. In the `POWHEG BOX V2` it is possible to specify if a parton arises from a resonance decay, and it is also possible to include radiative corrections to decaying resonances.

- **Photon radiation.** Radiation of photons can be treated now in the `POWHEG` framework. When an appropriate flag is turned on, the `POWHEG BOX` detects collinear configurations formed by electrically charged partons (including leptons) and photons.[1] Electromagnetic corrections are then included with the same formalism used for strong corrections. Photon radiation is computed with the same Shower type algorithm as strong radiation, and the hardest radiation can be also a photonic one when this feature is turned on.

- **Collinear singularities with massive fermion.** It is now possible to treat the collinear radiation from a massive fermion with the Shower Monte Carlo technique. In the `V1`, heavy quarks were always treated as heavy, i.e. the collinear singular limit for heavy quark radiation was not considered. In the case of charm and bottom, it may be appropriate to treat the heavy flavour as a light one, and to resum collinear singularities arising from it.

- **The MiNLO feature.** In processes that include singular associated production of jets, the MiNLO settings for the scale choice and the exponentiation of Sudakov form factors associated with singular Born kinematic configurations [1, 2] is available

[1]At the moment, photon splitting into charged particle-antiparticle pairs are not considered.

in `V2`. The MiNLO feature is useful when the Born configuration has collinear singularities, as, for example, in the production of an electroweak boson (a W, Z or H particle) in association with one or more jets. In these cases, the renormalization and factorization scales are chosen according to the CKKW [3] procedure, and large Sudakov double and single logarithms are exponentiated. It turns out that, when the MiNLO procedure is active, also the inclusive cross section obtained by integrating out the singular Born configurations is finite, and generally, it is accurate at the Born level. In simple cases of massive colourless objects produced in association with a jet, the MiNLO procedure can be tuned to yield NLO accuracy [2].

- **Reweighting.** With this feature it is possible to generate event samples with a set of weights associated with each event. The weights may correspond to any variation of the parameters of the run, as long as they don't affect the generation of phase space. Typical set of weights may involve renormalization and factorization scale variation, or PDFs variations. But one may also consider other variations of the parameters or settings, as long as they don't affect the generation of the Born phase space. For example, one can generate a sample of events with weights corresponding to the MiNLO feature being turned on or off, or one may consider variations of coupling constants and masses. In this last case, reweighting is possible only if the Born phase space does not depend upon the masses being varied. Thus, for example, if one would like to vary slightly the mass of a resonance, and the mass appears in the Born phase for importance sampling reasons, it is convenient to use different variables for the real mass and for the mass parameter used in the phase space calculation. Maintaining the latter fixed, one can generate weights for an event sample by varying the physical mass.

  The weights were originally specified in a `POWHEG BOX` specific format, using `#` commented lines in the LHE file. Furthermore, the Les Houches weights specifiers reported in `http://phystev.in2p3.fr/wiki/2013:groups:tools_lheextension` is also available.

  Depending upon one's analysis framework, one can make use of reweighting information in the way one prefers. In the `POWHEG BOX`, we have our own analysis framework, that we use to test our code and to perform phenomenological analysis. In `V2` we have extended this framework in order to efficiently make use of reweighting information. The new histogramming package `pwhg_bookhist-multi.f`, while being largely compatible with the previous histogramming package, can generate histogram data for each weight in an LHE sample in a single analysis run.

- **Fast underlying Born generation.** Event generation in `POWHEG` takes places in two steps. In the first step, an underlying Born configuration is generated, with a cross section proportional to the so called $\bar{B}$ function, that is the NLO cross section differential in the underlying Born phase space variables. In order to do so, the `POWHEG BOX` computes an upper bounding envelope of the so called $\tilde{B}$ function, which is a combination of the Born, virtual and real cross section. The upper bounding envelope consists of the product of step functions, each one of them function of a single integration variable. The efficiency of such upper bounding envelope is often quite small. In `V2`, there is the option of using an alternative strategy. Two upper bounding envelopes are constructed, one on the $\tilde{B}$ cross section, and the other on the ratio of the $\tilde{B}$ function divided by the Born cross section. For the generation of the underlying Born configuration, one first generates a configuration with the probability distribution given by the upper bounding envelope of $\tilde{B}$. By the usual hit and miss technique, one should then generate a uniform random number between zero and the value of the upper bounding envelope at the given point. If the number is below the value of the $\tilde{B}$ function at that point, the event is accepted, otherwise it is rejected. Before computing the $\tilde{B}$ function, which is a potentially CPU consuming procedure, one first computes the value of the Born term, and multiplies it by the upper bound of the ratio of the $\tilde{B}$ function over the Born term. This product is an upper bound on the value of $\tilde{B}$. If the generated random number is above it, it is certainly also above the $\tilde{B}$ function, and the event can be discarded without calling the $\tilde{B}$ function. This procedure is implemented with another improvement on the generation of the underlying Born configuration that allows a better parallelization of the step of grid formation.

Besides these point, a number of minor new features are also available:

- **Doubling the singular regions for fermionic emitters.** In `V1`, the splitting of a fermion into a fermion plus a gluon, was treated by always assuming that the gluon was the emitted parton. Since the transverse momentum of the splitting is defined as the transverse momentum of the emitted parton with respect to the emitter, this yields to the correct definition of the transverse momentum when the gluon is soft. According to this definition, configurations where the fermion has small transverse momentum but a large angle relative to the gluon will turn out to have large transverse momentum. These configurations were suspected to yield unphysical effects, and thus, in `V2`, it is possible to separate these collinear regions into two singular terms, one where the fermion is the emitter (damped when the fermion becomes soft) and one where the gluon is the emitted, that is damped when the gluon is soft.

- **Improving the upper bounds for the generation of radiation.** In the `POWHEG BOX` the hardest radiation is generated with a veto algorithm, that begins by generating radiation according to an upper bound of the ratio R/B (i.e. the ratio of the real cross section at a given underlying Born configuration over the Born cross section). The upper bound is given by a standard function of the radiation variables, times a normalization factor, that is computed for each subprocess of the real cross section. In complex processes, there is a large number of such subprocesses, and in order to get reliable normalization factors one would need to use a large number of sample points (the `nubounds` parameter in the `powheg.input` file). In `V2`, if the parameter `evenmaxrat` is set to 1 in the `powheg.input` file, the norms are combined (i.e. the maximum is taken) for all subprocesses that have matrix elements that are numerically identical.

- **Improving the separation of singular regions.** In ref. [4] we discussed an improvement on the separation of singular regions in the real cross section, that is useful in cases when the underlying Born configuration may develop singularities. This feature is turned on by default in `V2`. It is still possible to switch it off setting the flag `olddij` to 1 in the `powheg.input` file, in order to reproduce the old behaviour in `V1`. This is useful when porting a `V1` process to `V2`, if one wants to check that the ported program gives identical results, or if one suspects that the `V2` version has visible differences with respect to the `V1` version due to this feature.

2. Treatment of resonances

In the `POWHEG BOX V2` it is possible to treat production phenomena including radiation decay in a proper way, even when the decay products include coloured partons. In `V1`, the code had no knowledge of the origin of final state partons, and made no distinctions between directly produced partons, and partons arising from resonance decay. Consider, for example, the process of W production where the W decays into a $q\bar{q}$. The flavour structures of the Born process are represented in `V1` by an array of four integers, the first two giving the particle id of the incoming partons, and the last two those of the final state partons. For example, `[1,-2,3,-4]` means an incoming d and $\bar{u}$ quarks, and outgoing s and $\bar{c}$ quarks. A corresponding real graph including one gluon emission is has the flavour structure `[1,-2,3,-4,0]`, where 0 represents the gluon.[2] In `V1` the program will find three singular regions, one where the gluon becomes collinear to either incoming partons, and two with the gluon collinear to either final state partons. It will separate the real matrix element into the sum of three terms, each of them obtained by multiplying the full matrix element by an S factor, as described in Section 4.7 of ref. [5]. The real cross section is thus parted into components that are only singular in one singular region. In this way, however, the radiation from the resonance and from the final state is not correctly separated, and this will cause a distortion in the mass spectrum of W decay. In order to further clarify this problem, assume that part of an initial state radiation contribution (ISR from now on) is attributed to W radiation. This will result in an increased apparent mass of the W. On the other hand, if part of the W radiation is instead attributed to ISR, the apparent W mass will be decreased. Of course, at the end the total real cross section at order α_S will be reproduced correctly. However, when completing the event with the shower, the resonance mass will not be preserved correctly, since the resonance decay products are not correctly assigned already at the LHE level.

The problem of resonance mass distortion becomes more serious for more complex processes, where there are more final state partons besides the resonance decay products, or if more resonances are produced. The kinematic mapping used by `POWHEG` to construct the underlying Born configuration is such that the CM energy and the mass of the system

[2]In `POWHEG` the gluon id is 0 rather than 21, since this allows for simpler internal code.

recoiling against the splitting pair are conserved. On the other hand, in resonance decays, it is the four-momentum of the resonance that should instead be kept invariant. Because of these problems, radiation from resonances decays where never included in `V1`. When resonances decaying into coloured partons where needed, they where either handled approximately according to the prescription of ref. [6], or they were dealt with by substituting temporarily the id code of the decay products with different codes, so that the program would not recognized them as coloured partons.

The `V2` program can deal with decaying resonances in the narrow width approximation, provided that the process dependent code complies with the following requirements:

- The subprocesses for the real emission should also include as separate contributions the emission from the resonances.
- The code should provide the Born, Virtual and real matrix elements depending not only upon the initial and final state particles, but also as a function of the structure of the resonance decay cascade that has led to the given final state. In other words, real emission contributions arising from the decay products of the resonances and from the production process should be separated. In case of coloured resonances (like in top production) this separation is well defined in the narrow width approximation (see for example ref. [**?**]).
- The soft and collinear limit of the real cross section should match exactly those that can be computed using the standard eikonal and collinear approximation from the corresponding Born term *i*n the narrow-width limit. This is necessary, since the soft and collinear subtractions are computed using such approximation.
- The virtual corrections to the resonance's decays should be included.

The resonance decay feature of the `V2` can also be used without the inclusion of NLO corrections to resonance decays. In order to do this, it is enough not to include real radiation processes where the radiated parton belongs to the resonance.

As of now, the `V2` can thus treat resonances in the narrow width approximation up to NLO order. Nested decay structure are also allowed with no restriction. However, no provisions are given for the treatment of finite width interference effects. These, in fact, cannot be easily handled in a shower Monte Carlo framework, since there one must always respect a cascade structure, with the four momentum of the decaying object being preserved by the shower. On the other hand, it is conceivable that one can use matrix elements including interference, and that the assignment of components of the cross section to a given resonance decay cascade is performed in an approximate sense. There are several options for doing this in a general way. This topic is subject to further study, and will presumably be addressed in the future evolution of the `V2` code.

A further limitation of the present implementation of resonance decays is that the resonance decay cascade structure should be fixed at the Born level, in a sense that will become clear in the following subsection.

2.1 Implementation of resonances decays

In order to treat resonance decay cascades, the decay structure should be properly represented. In order to do so, the particles lists for each subprocess flavour structure should not only include the final state particles, but also the intermediate resonances. Furthermore, for each final state particle we should also store its mother resonance, if any. The array `flst_bornres(1:nlegborn,flst_nborn)` serves this purpose. Its entries are zero for the initial state particles and for the particles directly produced in the reaction, while, for particles coming from the decay of a resonance they are equal to the position of the resonance in the `flst_bornres(1:nlegborn,flst_nborn)` array. The array is initialized to zero, so that for processes not involving resonance decays everything works as in `V1`.

An instructive example of resonance decay code is given in the `ttb_dec` user processes, corresponding to $t\bar{t}$ production with decays. The flavour lists for the $gg \to (t \to (W^+ \to e^+\nu_e)\ b)\ (\bar{t} \to (W^- \to \mu^-\bar{\nu}_\mu)\ \bar{b})$ subprocess is:

```
flst_born(1:nlegborn,j)   =[ 0, 0, 6, -6, 24,-24,-11, 12, 13,-14, 5, -5]
flst_bornres(1:nlegborn,j)=[ 0, 0, 0,  0,  3,  4,  5,  5,  6,  6, 3,  4]
```

where the `flst_bornres` entry for t and $\bar{t}$ is zero (they come from the production process), for W^+ and b it is 3 (they come from the decay of the top, which is the $3^{\rm rd}$ entry) for the e^+ and the ν_e it is 5 (they come from the W^+, which is the $5^{\rm th}$ entry), and so on.

Notice that, although the part of the program that deals with flavour structures and determines the singular regions is fully general, the implementation in the rest of the program assumes that the resonance decay structure is the same for all Born subprocesses. This is because the Born phase space is one and the same for all Born subprocesses, and this cannot be the case if different resonance decay structures are present. Thus, the `flst_bornres(1:nlegborn,j)` array is in fact constant in the index `j`.

The array `flst_realres` serves the same purpose as the `flst_bornres` array, for the real emission graph. In this case, the radiated parton (the last in the particle list) can either arise from production (`realres(nlegreal,j)` entry equal to zero) or from a resonance (`realres(nlegreal,j)` entry equal to the resonance position). Thus, in this case, the `realres(1:nlegreal,j)` array is not constant in `j`.

The `BOX`, on the basis of the provided lists, sets up the parameter `flst_nreson` equal to the number of resonances that can radiate. This includes radiation from production, that is treated as a fictitious resonance, indexed by 0. Furthermore, the array `flst_reslist` is set up, that contains the index of each resonance that can radiate. Its first entry is always 0, corresponding to radiation in production. If there are no radiating resonances, the BOX sets `flst_nreson` to 1, `flst_reslist(1)=0`.

The user must provide a `setreal` routine that returns the matrix element for radiation of a specific resonance. In order to do so, the `setreal` routine must inspect the variable `kn_resemitter`, which is a pointer to the resonance that is radiating, and supply the corresponding real matrix element. The `setvirtual` routine should provide the sum of the virtual contributions for the production and decay processes, including corrections associated with resonance decays if one wishes to include them.

2.2 Internal implementation

In order to include radiation from decaying resonances, the POWHEG BOX program has been modified in three areas:

- The subroutines that deal with flavour structure and that find the singular regions (all subroutines in the file find_regions.f) need to be made aware of the resonance structure of the process.

- The subroutine that computes the real phase space for radiation (in file gen_real_phsp.f). The real phase space is built on a given Born phase space, as a function of three variables (kn_csi, kn_y and kn_phi). The algorithm for the construction of the real final state in V1 is described in [5, 7]. The algorithm preserves the CM energy and the mass of the system recoiling against the emitter and radiated parton. In order to deal with resonance decays, the final state radiation algorithm must instead preserve the resonance four momentum.

- The subroutines that compute soft and collinear limits in final state radiation (in file sigcollsoft.f). These subroutine must be made aware of the fact that if the radiation comes from a resonance the soft and collinear limits must be computed consistently with the construction of the real phase space in case of radiation from resonances. Furthermore, soft radiation from coloured resonances decays must be computed by considering as external legs the resonance itself (if coloured) and all its coloured direct sons. This also includes the case of a resonance decaying into another coloured resonance. The eikonal formula for soft radiation should be applied in this case by considering this resonance as a final state, on shell coloured particle.

- The subroutine btildevirt, in the sigsoftvirt.f file, should be suitably modified. In fact, this subroutine computes the analytic integral of the soft and collinear subtractions to the real cross section. This soft integral is performed in the CM frame, in the case of radiation arising from the hard process. On the other hand, the integral should be performed in the resonance rest frame if the soft radiation comes from a decaying resonance.

In the following we give some guidelines to understand the code modifications that were carried out in order to meet the above requirements.

2.2.1 Finding the singular regions

The subroutines for finding the singular regions are collected in the file find_regions.f. The most apparent difference in V2 with respect to V1 is that now the singular regions are determined from three arrays: the array of flavour structures, the array of resonance pointers, and the array of tags. We remind the reader that tags were introduced in V1 to mark the id of external particles in such a way that particles with the same id but different tags are treated as being different by the POWHEG code. This feature is used, for example, in VBF Higgs production, where it is convenient to mark the positive and negative rapidity partons that radiate the vector boson as being different particles. In `V1` tags were handled by mapping the id of tagged objects to a temporary new id, in such a way to as to induce the program to treat flavours with identical id but different tags as if they were different. This rather cumbersome method has been abandoned in `V2`. Now, at the heart of the procedure for finding singular regions, there is the subroutine `same_splitting(a,ares,atags,indexreal,i,j,ibornfl,itag,iret)`. This subroutine is called with the first three arguments equal to the flavour structures of the real graphs (typically `flst_real`), the second argument is the array of resonance pointers (typically `flst_realres`) and the third argument is the the array of tags (typically `flst_realtags`); `indexreal` identifies which real flavour structures (i.e. it is the second index in the three arrays `flst_realres`, and `flst_realres` and `flst_realtags`), `i` and `j` identify the pair of partons being considered for collinear singularities. If the two parton may give rise to a collinear singularity (like for example, a fermion antifermion pair of opposite flavours, arising from the same resonance and having the same tag) the `iret` flag is set to 1 on return, `ibornfl` is set to the flavour of the parent parton in the splitting, and itag is set to its tag value. Otherwise, `iret` is set to -1 upon return.

At variance with the `V1` code, in the `V2` code all properties of the flavour lists require also consistency of the resonance assignment. For example, the subroutines `flavequivl` and `flavequivr`, that check if two flavour structure are equivalent up to a permutation of their entries, require as argument also the array of resonances pointers and tag pointers, and check that also these arrays are equivalent under the same permutation. Equivalence of flavour structures should also account for the fact that resonance decay products may appear in different order in the flavour lists being compared. This requires the use of a recursive function (the `rec_ident` function) in order to perform the comparison in a clean way. The `flavequivl` and `flavequivr` are used to determine if two singular flavour configurations are equivalent, and should be combined. It is instructive to illustrate these two functions here, to give an idea about how the code that finds the singular regions in `V2` works. First of all, we have

```
logical flavequivl(m,n,ja,jb,arr,arrres,arrtags)
m = dimension of flavour lists
arr(m,*)       flavour list
arrres(m,*)    resonance list
arrtags(m,*)   tag list being \comment{GZ: ?}
ja, jb         two indeces
returns true if
[arr(1:n,ja),arrres(1:n,ja),arrtags(1:n,ja)] is equivalent to
[arr(1:n,jb),arrres(1:n,jb),arrtags(1:n,jb)]
up to a permutation of the flavour entries.
```

By inspecting the code, we see that `flavequivl` invokes `flavequivr` :

```
logical flavequivr(n,a,ares,atags,b,bres,btags)
n = length of flavour list
```

```
a(n), ares(n), atags(n), first flavour structure
b(n), bres(n), btags(n), second flavour structure
```

By inspecting the code, we see that flavour and tags of the first two entries must be identical (they correspond to initial state partons, and cannot be permuted). The code, for each flavour entry in the `a` array that has zero `ares` (i.e. only particles produced in the hard interaction), finds an equivalent particle in the `b` array. Equivalence is determined by the recursive function `rec_ident`, that requires that if the particle being considered is a resonance, it should have an equivalent decay chain, where again the equivalent means up to permutations of the decay products. The function

```
recursive function rec_ident(n,ia,ib,a,ares,atags,b,bres,btags)
```

checks if the entries `ia` and `ib` in the `a` and `b` arrays are equivalent up to a permutation. By inspecting the code, we see that first of all the two entries should have the same flavour and tags, otherwise `.false.` is returned. If neither `ia` in `a` or `ib` in `b` have decay products, the function returns `.true.`, and if they have a different number of decay products, it returns`.false.`. Then the code goes through all particles that are sons of the `ia` entry, and seeks an equivalent entry among the sons of `ib` in the `b` array. Equivalence is tested for calling `rec_ident` recursively. If for a decay product in `a` no decay products are found in `b`, the function returns `.false.`. Similarly, if at the end there are unmatched decay products in `b`, the function returns `.false.`.

2.2.2 Real phase space

The machinery for resonance radiation requires a considerable enhancement of the final state real radiation phase space routines. First of all, besides initial and final state partons, also resonances appear in the flavour list, and their kinematics must be properly set. Furthermore, radiation from a resonance implies that the four-momentum of the resonance should be preserved. In `V2`, the phase space mapping for radiation is the same as in `V1`, but, in case of radiation from a resonance, it is applied only to the resonance decay products in the resonance rest frame. This can be seen in the `gen_real_phsp_fsr_rad` routine. At the very beginning, the code checks if the `flst_bornres(kn_emitter,1)` is different from zero (we remind the reader that the `flst_bornres` array is independent upon the second index). If this is the case, the momenta of all resonance decay products (including the sons of resonances) are copied into an array, and boosted to the resonance rest frame. The routine that builds the radiation phase space is called using this array. After this is done, an opposite boost is performed, and the momenta are copied into the `kn_real` array.

The kinematic functions that are used for the separation of the real cross section into the various singular contributions (i.e. the `kn_dijterm` arrays) are frame dependent, and in resonance decays they are better defined in the resonance CM frame. The `compdij` routines in `V2` are thus modified accordingly.

As will be better discussed in the following, in `V2` there is an improved separation of the singular regions, that no longer uses the `kn_dijterm` arrays. The relevant routines (in the file `ubprojections.f`) are also aware of resonances, and are consistent with resonance decays.

2.2.3 Collinear and soft limit of the real cross section

The routines that compute the final state collinear and soft limits of the real cross section are modified in `V2` to account for resonance decays. The collinear and soft limits are in fact defined by taking the appropriate limit of the radiation variables `kn_y`, `kn_csi` and `kn_azi`, i.e. taking `kn_y` to ± 1 at fixed `kn_csi` and `kn_azi` for the collinear limits, and taking `kn_csi` to 0 at fixed `kn_y`, `kn_azi` for the soft limit, keeping fixed the underlying Born kinematics. Since the phase space construction as a function of the radiation variables is different in case of resonance decay, it is clear that also the soft and collinear limits are different, and should reflect the modified kinematic construction of the real phase space when resonance decays are considered.

In particular, the `buildfsrvars` routine, that builds final state radiation variables in the soft limit, now, in case of resonance decays, performs a boost to the resonance rest frame before computing the radiation variables.

Particular attention is taken when computing the soft limit of a real amplitude. Here, if the radiation comes from a resonance, one must consider all on shell particles involved in the resonance decay as possible emitters of soft gluons. These involve also the incoming resonance itself. The `colcorr(j,iub,res)` function, in the `sigcollsoft.f` file, checks if particle `j`, in the underlying Born configuration labeled `iub`, can give rise to soft eikonal terms in the decay of the resonance `res`. The function simply checks if the particle is coloured, if it arises directly from the resonance being considered, or if it is the resonance itself. The eikonal terms depend upon the direction of the soft emitted particles. This is stored in the `kn_softvec` vector by the phase space routine `setsoftvecfsr`, that must now determine the direction of the soft vector accounting for the fact that it may originate from a resonance decay. If the emitter being considered belongs to a resonance, its four-momentum is boosted to the resonance rest frame. The soft vector is then constructed based upon the boosted emitter four-momentum, and at the end of the construction the soft vector is boosted back to the partonic CM frame.

2.2.4 Integrated soft corrections

The soft counterterms that are subtracted to the real cross section according to the subtraction method, are integrated analytically and added back to the soft-virtual contributions to the NLO cross section.

We remind the reader that, in the narrow width approximation, virtual and soft collinear corrections factorize among production and decay [?]. In particular, soft corrections for a decaying resonance are obtained by considering all pairs of coloured particles involved in the calculation of the decay. If the resonance is coloured, eikonal terms associated with the resonance emitting a soft gluon will also be included.

In the `POWHEG BOX`, soft corrections are computed in the `btildevirt` subroutine, in the `sigsoftvirt.f` file. Since the soft counterterms to the real cross section are computed in the resonance rest frame in case of emission from a resonance, the corresponding integrated soft contributions must also be computed in the resonance frame, and thus the `btildevirt` subroutine must be modified to be made consistent with the resonance decay options. The subroutine computes all soft contributions together. In the `V2` code, it loops over all resonances for which radiative corrections are included. This is apparent in the `btildevirt` code, where a loop `do jres=1,flst_nreson` is clearly visible. The underlying Born momenta are computed by boosting the `kn_cmpborn` variables to the resonance rest frame. The loop over final state massless partons, that accounts for the inclusion of final state collinear remnants, and over pairs of final state coloured partons, that accounts for soft eikonal contributions, is restricted to the resonance itself (if coloured) and to its direct coloured decay products. These are typically selected by cycling over particles not satisfying these requirements, with statements of the form `if(leg.ne.kres.and.flst_bornres(leg,jb).ne.kres) cycle`.

3. The fastbtlbound feature

This feature was introduced to reduce the number of calls to the virtual routines, for processes where the virtual corrections are particularly demanding.

The `POWHEG BOX` generates the underlying Born and radiation kinematics using a hit and miss technique. An upper bounding envelope of the $\tilde{B}$ function is found, of the form

$$\tilde{B}(X) \leq \prod_{i=1}^{n} f^{(i)}(X_i), \tag{3.1}$$

where the X_i are the integration variables, and the $f^{(i)}$ functions are step functions of the integration variables. The size of the step is determined by the importance sampling grid itself, as documented in ref. [8]. In order to generate a configuration, the points X_i are first generated with a probability distribution equal to $f^{(i)}(X_i)$. Then a uniform random number r, with

$$0 \leq r \leq \prod_{i=1}^{n} f^{(i)}(X_i) \tag{3.2}$$

is generated. One then computes $\tilde{B}(X)$. If $r \leq \tilde{B}(X)$ we have a hit, and the configuration is kept. Otherwise the configuration is rejected (we have a miss), and we restart the procedure.

It is clear that, if the number of integration variables is large (as in our case), an upper bound of the form eq. (3.1) will generally be highly inefficient, just because the product of a large number of terms will tend to build up large values. In order to remedy to this problem, we have exploited the fact that the $\tilde{B}$ function is equal to the Born cross section plus higher order terms. It is thus natural to expect that an upper bound of the form

$$\tilde{B}(X) \leq B(X) \times \prod_{i=1}^{n} g^{(i)}(X_i), \tag{3.3}$$

will be more adequate. We thus determine also the g functions for this bound, using the same technique used for the f functions. The generation of the underlying Born phase space is then modified as follows. Before computing $\tilde{B}$ in order to test for a hit or miss, we compute the right hand side of eq. (3.3). If it is smaller than r, $\tilde{B}$ will also be smaller than

r, and we thus know that we have a miss without the need to compute the time consuming $\tilde{B}$ function.

This modified procedure for the generation of the underlying Born configuration is activated by inserting the line `fastbtlbound 1` in the `powheg.input` file. This triggers the setting of the internal flag `flg_fastbtlbound=.true.`. By inspecting the integrator routines (typically `mint` and `gen` in the `integrator.f` file), comparing `V2` with respect to `V1`, we see that, besides the array `ymax(nintervals,ndim)`, also an array `ymaxrat(nintervals,ndim)` appears. While the `ymax` array stores the values of the $f^{(i)}$ step functions at each step and for each integration dimension, the `ymaxrat` array stores the $g^{(i)}$ step functions.

In order to implement the `fastbtlbound` feature, the (already complex) call sequence of the function to be integrated had to be modified even further. Here we summarize how the call sequence works, also reminding the reader of the reasons for such a cumbersome structure. The function to be integrated `fun` is invoked as follows

```
ifun = fun(x, vol, ifirst, imode, vfun, fvun0)
double precision x(ndim)   ! integration point
double precision vol       ! phase space volume of the cell being considered
integer ifirst             ! flag used to implement the "folding" feature
integer imode              ! if 0, compute only the Born contribution vfun0
                           ! if 1, compute also the full result vfun
double precision vfun      ! output value
double precision vfun0     ! approximate output value
```

The flag `ifirst` was already present in `V1`, and it was used to implement the "folding" feature. When called with `ifirst=0` the function computes all contributions (Born, virtual, real and collinear remnants). When called with `ifirst=1`, only the real and collinear remnant contributions are computed, and it is assumed that Born and Virtual are the same as in the previous call. When called with ifirst=2, nothing is computed, and the accumulated values of the previous calls (up to the last call with `ifirst=0`) are returned. If the integer return value `iret` is 0 upon exit, it is assumed that the function can compute the approximate value `vfun0` (which can be computed much faster than `f`), while if it differs from zero it is assumed that no approximate value is provided, and the `fastbtlbound 1` cannot be used. This feature is necessary, because even if the `flg_fastbtlbound` is set to true, the feature cannot be used when computing the remnant cross section. The `sigremnant` function is thus set up to always yield a return value of 1.

4. The storemintupb feature

The upper bound step functions $f^{(i)}$ and eventually $g^{(i)}$ are normally determined with a simple algorithm. During the stage of the computation of the NLO cross section, if the computed value of the function at the given point is above the value of the f (or g) function in the corresponding cell of the integration volume, the value of each of the $f^{(i)}$ function corresponding to that cell is increased by a factor `1+0.1d0/ndim`, in such a way that the value of the $f^{(i)}$ function in the cell increases roughly by 10%. This is done directly in the integration routine `mint`, as soon as a value for the function is made available. The method is not guaranteed to converge to a useful upper bound. One must check at the end of the run that the number of upper bound failures for the determination of the underlying Born kinematics is small, typically below 1% of the total number of generated events. If this is not the case, the number `ncall2*itmx2` must be increased. This method is clearly somewhat limited, since it certainly does not seem to make wise use of the calculated values of the function to be integrated. Especially for the case of the `btilde` function, these values may be quite expensive, in terms of computing time. In `V2` an alternative method is also proposed, that is activated by the flag `storemintupb 1` in the `powheg.input` file. When this method is activated, the result of each function call is stored in a file together with the coordinates of the cell where the integration point was found. At the end of the NLO calculation stage, before the computation of the upper bounds for the generation of radiation, this file is read into memory, and an upper bounding envelope is computed with a more elaborate algorithm. The disadvantage of this step is that a large memory array must be allocated for this operation, that must contain all the function calls performed for the NLO calculation. On the other hand, the array is allocated dynamically, and it is deallocated as soon as the calculation of the upper bounding envelope is completed. Thus, the large memory usage lasts only for a small fraction of the total computing time.

The implementation of the `storemintupb` feature can be found in the file `mint_upb.f`. The integrator routine `mint` has also been modified to implement this feature. If the `storemintupb` feature is set, the internal logical flag `flg_storemintupb` is set to true. If this flag is set, the initialization routine `bbinit` calls the subroutine

```
startstoremintupb(filetag),
```

where `filetag` is a string that will appear in the name of the file that will store the results of the function calls, before calling the integrator. The tag is `'btildeupb'` for the btilde function, and `'remnupb'` for the remnant function. The name of the file is obtained by concatenating the prefix of the run (default `pwg`), the tag, and, if a parallel run is being performed, the four digits integer id of the run (i.e. `0001`, `0002`, etc.). After the initialization routine is called, the integrator `mint` can be invoked. By inspecting the integrator code (in `integrator.f`), we see that, if the flag `flg_storemintupb` is set to true, rather than adjusting directly the upper bounding envelope, the routine calls

```
storemintupb(ndim,ncell,imode,f,vfun0)
integer ndim                  ! dimension of the integral
integer ncell(ndim)           ! integer coordinate of the cell where
                              ! the integration point lies
integer imode                 ! 0 (no fastbtlbound) 1 (with fastbtlbound)
double precision f            ! value of the function
double precision vfun0        ! value of the approximate function
                              ! (typically the Born approximation)
```

The `cell` array is stored in the file in an ascii format, each entry as a two digit integer, without spaces among each other, while the `f` (and eventually also the `vfun0`) values are stored as 11 digit floating point numbers. This format is adequate, since the number of cells for each coordinate is 50. At the end of the integration, the `bbinit` subroutine calls the `stopstoremintupb` subroutine that closes the output file.

The `loadmintupb` subroutine actually computes the $f^{(i)}$ (and eventually the $g^{(i)}$) step function, that are stored in the `ymax` and `ymaxrat` arrays). Much of the complexity in this subroutine is due to the fact that, in case of parallel runs, it must be able to load the stored function call results from several files carrying all possible four digit integer id's. The calling sequence is

```
loadmintupb(ndim,filetag,ymax,ymaxrat)
integer ndim                         ! dimensionality of the integrand
character *(*) filetag               ! tag in name of file
double precision ymax(50,ndim)       ! output array for the upper
                                     ! bounding envelope
double precision ymaxrat(50,ndim)    ! auxiliary array if the
                                     ! fastbtlbound feature is being used.
```

This subroutine loads the file `pwgprefix//filetag//'.dat'` containing the results of the function calls, and computes the `ymax` and `ymaxrat` arrays. If the `manyseeds` option for parallel runs is active (and the internal character string `rnd_cwhichseed` is set as a consequence to a four-digit integer id), all files with names of the form `pwgprefix//filetag//'-'//id//'.dat'`, with `id` any four digit string (left padded with zeros) will be loaded. If `filetag='btildeupb'` and `flg_fastbtlbound=.true.` the subroutine assumes that both `ymax` and `ymaxrat` arrays should be computed, and thus that the files contain two floating point numbers per line. Otherwise a single floating point number per line is read from the file, and only the `ymax` array is computed. The program calls the subroutine

```
getlinemintupb1(filetag,ndim,cells,f,f0,iret)
```

The return flag `iret` is set to 0 normally, and is set to 1 upon end of data. Subsequent calls restart from the beginning of the data set. The subroutine works by counting the lines in the files, allocating memory for them, and reading them into memory upon the first call (we will not describe the working of this feature here, reading the code should clarify how this works). All subsequent calls use the data stored in memory. A special call with `iret=-10` causes the deallocation of the memory arrays.

The algorithm for the computation of the step functions $f^{(i)}$ and $g^{(i)}$ works as follows. Initially the average values of `f` and (eventually) `f0` are computed, and the `ymax` and `ymaxrat` arrays are set to these average values. Then the program loops over the data set. If for a cell one finds that the product of the step functions, computed as

```
prod=1
do kdim=1,ndim
   prod=prod*ymax(cells(kdim),kdim)
enddo
```

(and similarly for `ymaxrat`) is smaller than `f`, each of the values of the step functions in the corresponding cell is increased as follows

```
            do kdim=1,ndim
               ymax(cells(kdim),kdim)=
     1              ymax(cells(kdim),kdim)*(f/prod+0.1)**(0.01/ndim)
            enddo
```

This guarantees an increase in the `prod` value of `(f/prod+0.1)**0.01`. This increment is insufficient to yield `f<prod`, and we can estimate that we would need about 100 sweeps through the data set to reach this point. The program does in fact sweeps through the data set several time, accumulating the total failure value (i.e. summing up the `f-prod` values). As long as the total failure value divided by the total sum of the `f` values is above 1/1000, the program continues to sweep through the data set. Messages reporting the ratio of the failure value over the total are printed to the standard output for each sweep. Once the failure value is sufficiently small, the program reports some statistics and returns.

The computed values of the `ymax` and `ymaxrat` arrays are stored in grid files named `prefix//'fullgrid'.dat`. In case the `manyseeds` option is active, these files will also carry the corresponding four digit identifier. In this last case all these files will be identical, and in fact only one of them would be needed. On the other hand, when doing parallel runs we cannot make any assumption about our hardware setup (a multi-node workstation, a batch system, etc.), so we need to prevent different instance of the program writing to the same file. Maintaining different file names for identical files seems to be the safest solution. If one needs to run the program again using the grids that are already present, only the `fullgrid` files will be read in, and the `loadmintupb` step will be skipped.

5. Parallel runs

It is possible to split a `POWHEG` run into several parallel ones. This is obviously the case at the stage of event generation, since same instances of the program run with different random seeds will obviously lead to independent event samples. For very complex processes, however, also the computation of the NLO cross section, the calculation of the optimized integration grid, the upper bounding envelope for the generation of the underlying Born phase space configurations, and the normalization of the upper bounding function for the generation of radiation need to be computed in parallel. In order for the parallel runs to be useful, each set of runs should complete at each stage, in such a way that at the subsequent stage all results obtained at the previous stage can be assembled and used.

The parallel run features of `V2` is activated by the following flags:

```
manyseeds 1 ! get the seeds for the random number generator from the
            ! pwgseeds.dat.
parallelstage <m>  ! <m>=1...4, which level of parallel stage
xgriditeration <n> ! <n> is the iteration level for the calculation of the
                   ! importance sampling grid improvement (relevant only for
                   ! parallelstage=1
```

The pwgseeds.dat file contains a list of integer seeds, one per line. If the manyseeds flag is set, when the pwhg_main program starts, it asks for an integer number m. This number is used to select a line in the pwgseeds.dat file, and the integer found there is used to initialize the random number generator. In this way, several runs can be performed simultaneously in the same directory, all using different random number seeds. Furthermore, the integer m appears as a four digit integer identifier (with leading zeros) in the name of all files that are generated by the corresponding run.

The pwhg_main program runs in four stages:

1. An importance sampling grid for the integration of the inclusive cross section is built and stored. This stage is run when parallelstage=1. The number of calls to the inclusive cross section is controlled by the input variable ncall1. The variable itmx1 in the powheg.input file is ignored in this case, and is internally set to 1. If also a remnant cross section is present, the number of calls for the remnants is equal to ncall1, unless a parameter ncall1rm appears. Initially, this stage is called with xgriditeration=1. The information that is needed to compute the importance sampling grids are stored in the files named (assuming $m = 1$ and the default prefix pwg) pwggridinfo-btl-xg1-0001.dat for the $\tilde{B}$ function and pwggridinfo-rmn-xg1-0001.dat for the remnant. The quality of the grid for each single run is represented in the topdrawer files pwg-xg1-0001-btlgrid.top. These plots are not very significant, since what counts is the grid formed by assembling all the information contained in all the pwggridinfo files. These are generated in the subsequent step, are named pwg-0001-btlgrid.top, and are all equal among each other for all values of m that have been used.

   Observe that it is important that all parallelstage=1 runs complete, in order to use all the produced information for building the grid.

   If after the parallelstage=1 and xgriditeration=1 run the grids do not look smooth enough, one or more iterations with xgriditeration=2 and higher, can be attempted. At this stage, the relevant files are named pwggridinfo-btl-xg2-0001.dat, pwggridinfo-rmn-xg2-0001.dat and pwg-xg2-0001-btlgrid.top, and so on. Increasing ncall1 also helps in getting more satisfactory grids.

2. This stage is performed after the parallelstage=1 run, by setting parallelstage=2. The integral of the inclusive cross section is computed, and an upper bounding of the integrand, having the form of a multidimensional step function on the grid, is computed and stored. The number of calls used for this task is controlled by ncall2 and itmx2 (the total number of calls per run is ncall2*itmx2). Again, independent variables for the remnants are also available if needed, ncall2rm and itmx2rm. If the files with information on the importance sampling grid are missing, the program complains and stops. The integration and upper bounding envelope information is stored in files named pwggrid-0001.dat. If the storemintupb flag is set, auxiliary files named pwgbtildeupb-0001.dat and pwgremnupb-0001.dat are created and loaded with these information. Again, all processes run at this stage should be completed

before going on. Only at the next stage these files are all loaded and assembled to get the cross section and upper bounding envelopes using the full statistics of the runs. Files named `pwgfullgrid-0001.dat` are then created. They are all equal among each other. On subsequent runs, if any `pwgfullgrid-0001.dat` is present, it is loaded in place of all the others, since it contains all the necessary information.

3. This is obtained by setting `parallelstage=3`. The upper bounding factors for the generation of radiation are computed at this stage. The number of calls (per run) is controlled by the variable `nubound`. The computed factors are stored in the files named `pwgubound-0001.dat`.

4. With `parallelstage=4`, the program starts generating events, that are stored in files named `pwgevents-0001.lhe`. The number of events per file is controlled by the `numevts` parameter in the `powheg.input` file.

The behaviour of the program when the `parallelstage` variable is set, is rather different with respect to the default behaviour, as far as using previously existing files. In the standard behaviour, if the `use_old_grids` variable is set to 1, if old grid files are present they are loaded, and the stage of grid formation is skept. On the other hand, if the `parallelstage` variable is set, the program assumes that all the files needed to arrive at the given stage are present, and tries to load them. If it does not find what it needs it aborts with an error message. The current stage calculation is performed no matter whether old current stage files are present, and eventually present, old current stage files are overwritten.

When performing parallel runs, for very slow processes that require very many CPU's, the number of events per process may not be enough for building a valid upper bounding grid (see ref. [9]). In this case, it is convenient to set the flag `storemintupb=1`. In this way, the results of the calls to the inclusive cross section are all stored in files, and the upper bounding envelope is built at a later parallel stage, by reading and assembling all these files (see Sec. 4).

5.1 Files to inspect at the end of the run

The file `pwgstat.dat` (for single run), or `pwg-stY-0001-stat.dat`, with the stage number `Y` equal to 2 or 3, contain information on the value of the cross section. Notice that, in case of parallel runs, the stage 3 files are the final ones, since they refer to the result of the combination of all the stage 2 runs, while the stage 2 files only include the result of a single run. It is important to check that the cross section has the required precision.

At the end of the runs, the values of certain counters are printed in files named `pwgcounters.dat`, or `pwgcounters-stY-XXXX.dat`, where `Y` is the `parallelstage` value and `XXXX` stands for the four digits run number in case of parallel runs. It is important to check in the stage 4 files (`pwgcounters-st4-XXXX.dat`) that the number of upper bound violation in the generation of radiation and in the inclusive cross section are much smaller than the total number of events. They appears as follows:

```
upper bound failures in generation of radiation = 636.000000000000
upper bound failure in inclusive cross section = 340.0000000000000
```

These numbers should be of the order of a percent of the total number of events. If the number of failure for the inclusive cross section is too large, the number of calls `ncalls2` should be increased. If the failure in the generation of radiation is too large, then `nubound` should be increased. In this last case, one can also attempt to increase `xupbound`, with no need to rerun the stage of generation of the upper bounds for radiation (that is to say, stage 3).

The counter files also report the approximate value of the total number of seconds spent doing the virtual and real calculations. These numbers are significant only if the program is run at 100% speed. If the virtual time is large compared to the real time, it may be convenient to increase the folding parameters `foldcsi`, `foldy` and `foldphi`, that also reduces the fraction of negative weighted events (for the allowed values for these parameter see the `POWHEG BOX` manual).

5.2 Shower and analysis

The user can shower the Les Houches event with whatever software he likes. We provide elementary interfaces to `Pythia6` and `Pythia8`. An interface to `HERWIG` can be easily developed on the basis of the examples in other `POWHEG BOX` processes. An enhancement of the previous histogramming package in the `POWHEG BOX` does now output the histograms associated with different weights, if present in the Les Houches event file. Assuming that all event files are merged into a single `pwgevents.lhe` files, in order to run the builtin analysis, for e.g. the $W + 2$ jet process, one proceeds as follows:

```
$ cd POWHEG-BOX/W2jet
$ make main-PYTHIA-lhef
$ cd test-lhc
$ ../main-PYTHIA-lhef
```

where `test-lhc` is the directory where the events are stored. The output file is
`pwgPOWHEG+PYTHIA-output.top.`
If more weights are present in the event file, files of the form
`pwgPOWHEG+PYTHIA-output-WY.top`
are produced, where `Y` is the weight ordering number in the event file. These files are gnuplot data files, and can be plotted with the gnuplot program.

The analysis can also be run in parallel. If the program does not find a `pwgevents.lhe` file, it prompts for a filename. If the file name has the form `pwgevents-XXXX.lhe`, then the output files have the name
`pwgPOWHEG+PYTHIA-output-XXXX-WY.top`
This is useful if one wants to speed up the analysis of parallel runs. It is up to the user, at the end, to combine the files. A fortran program `mergedata.f` is provided for this purpose in the `Version-pre2-1` directory. Its use is self explanatory.

5.3 A typical example of a script to perform parallel runs

A typical script to perform a parallel run on a 48-core workstation is illustrated below. One first sets up a standard `powheg.input` file (called e.g. `powheg.input-save`), containing the line `manyseeds 1`, and a `pwgseeds.dat` file with at least 48 lines, each line containing a different integer to seed the random number generator. The following script can perform the parallel run:

```
# two stages of importance sampling grid calculation
cat powheg.input-save > powheg.input
echo "parallelstage 1" >> powheg.input
echo "xgriditeration 1" >> powheg.input
cp powheg.input powheg.input-1-1         # save a copy of the
                                         # powheg.input for this stage
for i in {1..48}
do
echo $i | ../pwhg_main > run-st1-xg1-$i.log 2>&1 &
done
wait

cat powheg.input-save > powheg.input
echo "parallelstage 1" >> powheg.input
echo "xgriditeration 2" >> powheg.input
cp powheg.input powheg.input-1-2         # save a copy of the
                                         # powheg.input for this stage
for i in {1..48}
do
echo $i | ../pwhg_main > run-st1-xg2-$i.log 2>&1 &
done
wait

# compute NLO and upper bounding envelope
# for the generation of the underlying born configurations
cat powheg.input-save > powheg.input
echo "parallelstage 2" >> powheg.input
cp powheg.input powheg.input-2         # save a copy of the
                                       # powheg.input for this stage
for i in {1..48}
do
echo $i | ../pwhg_main > run-st2-$i.log 2>&1 &
done
wait

# compute upper bounding coefficients for radiation
```

```
cat powheg.input-save > powheg.input
echo "parallelstage 3" >> powheg.input
cp powheg.input powheg.input-3         # save a copy of the
                                       # powheg.input for this stage
for i in {1..48}
do
echo $i | ../pwhg_main > run-st3-$i.log 2>&1 &
done
wait

# generate events
cat powheg.input-save > powheg.input
echo "parallelstage 4" >> powheg.input
cp powheg.input powheg.input-4         # save a copy of the
                                       # powheg.input for this stage

for i in {1..48}
do
echo $i | ../pwhg_main > run-st4-$i.log 2>&1 &
done
wait
```

6. Massive emitter

In `V1`, heavy flavours where either treated as very heavy, or they where treated as massless. No attempt was made to resum collinear singularities arising from the emission of a not-so-heavy quark. In `V2`, a feature was added that allows to treat the emission from a heavy quark line as a collinear singular emission. In order to turn on this feature, it is enough to put the final state heavy quark particles at or after the `flst_lightpart` index. We remind the reader that the `flst_lightpart` is the index of the first light parton appearing in the final state in the flavour configurations for the Born, virtual and real amplitudes. All partons starting from the `flst_lightpart` down to the end of the flavour list are assumed to be capable of forming collinear singularities when paired among each other, or when paired with initial state partons. The subroutines that find the singular regions look at all pairs of final state partons starting from the `flst_lightpart` one, as well as each single parton starting from the `flst_lightpart` and paired with an initial state parton, and check if a collinear singularity is possible.

We will illustrate with an example the behaviour of the generator when such feature is turned on. We will consider W production in association with a $b\bar{b}$ pair. A typical Born level configuration is given by `[  1, -2, 11,-12,  5, -5]`, corresponding to the annihilation of a down and an anti-up into a W^- decaying into an electron and an antineutrino, produced in association with a $b\bar{b}$ pair. A typical real emission graph is like `[  1, -2, 11,-12,  5, -5,  0]`, where an additional gluon is emitted. If the `flst_lightpart` is set to 7, only the emitted partons are considered for collinear singularities. All the generation of radiation will be computed as if it was coming from the initial state partons. Soft radiation from the final state $b, \bar{b}$ quarks will also be treated kinematically as arising from the initial state. We stress that this behaviour is perfectly consistent with NLO accuracy. However, if we have reasons to believe that the kinematics of heavy flavour emission is such that important Sudakov effects should be present, these Sudakov factors are not implemented in this case. If instead we set `flst_lightpart=5`, the b and $\bar{b}$ partons will also be considered candidates for the formation of collinear singularities. In particular, both the b and $\bar{b}$ partons can pair with the final state gluon to form a collinear pair. The corresponding underlying Born kinematic configuration is simply given by removing the gluon from the final state, and thus it exists. On the other hand, b and $\bar{b}$ can form a collinear pair among each other, thus merging into a gluon. But no Born graphs are given that do not have the $b\bar{b}$ pair in the final state, and thus no collinear region will be emitted for this configuration.

The handling of the collinear singularity due to a massive fermion-vector pair in the final state requires a dedicated modification of the phase space routine for FSR real emission. The kinematic mapping of a real emission graph with a gluon-heavy quark to a corresponding underlying Born configuration is performed in the same way as for a massless emitter. In the partonic CM system, we boost the recoil system (i.e. the system of all final state particle, excluding the heavy quark and the gluon) along the direction of its total 3-momentum, and fix the 3 momentum of the heavy quark so as to balance the recoil system. The boost is chosen in such a way that the partonic CM energy is conserved. In the generation of the real emission phase space, we start with an underlying Born configuration and three radiation variables, ξ, y and ϕ, and we have a prescription for the construction of the phase space. In the case of massive emitter, the meaning of the y radiation variable could not be preserved. The real phase space construction for a massive emitter has been described in full details in appendix A of ref. [10], and will not be duplicated here. We will, however, remind the choice for the transverse momentum of the emission is defined to be

$$K_{\perp}^{2} = \frac{k^{0}}{p^{0}} 2k \cdot p \,, \tag{6.1}$$

where k is the four momentum of the gluon and p is the four momentum of the lepton in the partonic CM frame. In the massless limit this becomes

$$\frac{k^{0}}{p^{0}} 2k \cdot p \approx 2(k^{0})^{2} \, (1 - \cos\theta), \tag{6.2}$$

where θ is the angle between k and p in the CM frame. It thus corresponds to the transverse momentum definition for final state singularities in the massless case. The specification of the transverse momentum, together with the definition of the mapping of a real radiation configuration to an underlying Born one is all one needs to specify the physics of radiation, since the POWHEG Sudakov form factor only depends upon these two ingredients.

6.1 Implementation in the code

The subroutines for final state radiation in the gen_real_phsp.f file, in order to implement the massive emitter construction, check explicitly if the emitter has non zero mass, using the kn_masses array. The subroutine that actually performs the real phase space construction, barradmapmv, is called in place of barradmap in case of a massive emitter.

The definition of the transverse momentum for radiation, in the subroutine pwhg_pt2 (in the gen_radiation.f file) now also checks for a massive emitter, and in such case calls the appropriate subroutines to compute the transverse momentum according to formula (6.1). The subroutine gen_rad_fsr, that computes the generation of final state radiation, and the subroutine that computes the upper bound for radiation (pwhg_upperb_rad), check now if a massive emitter is present, and, if so, implement the formulae described in ref. [10].

The collinear limits of the real amplitude are computed by the subroutines collected in the sigcollsoft.f. In case of a massive emitter, there is really no collinear singularity, and thus the collfsralr subroutine returns zero for a massive emitter. There can be, however, a soft component. This does not require any particular treatment, since the soft limit of the real amplitude does not explicitly depend upon the emitter, but only upon the radiated parton. The separation of the real amplitude in terms of the R_α components does of course depend upon the singular region, but this is only a matter of associating the appropriate S_α factor to the real amplitude. In the case of a massive emitter, the d_{ij} function is chosen equal to

$$d_{ij} = \left(2k_i \cdot k_j \times \frac{E_j}{E_i}\right) , \tag{6.3}$$

where i is the (massive) emitter and j is the (massless) emitted parton, computed in the partonic CM system, or in the resonance CM system if the emitter and emitted partons arise from the decay of a resonance.

7. Photon radiation

Some support for processes involving photon radiation is built in V2. It originates from the work done in ref. [10], where W production including electroweak corrections is considered. At present, the electromagnetic machinery developed in V2 is not complete, but a number of processes can already be addressed.

7.1 Finding the singular regions

First of all, the algorithm for finding singular regions in V2 is now capable of identifying singularities related to photon emission. The relevant code, in the find_regions.f file, can be found in the same_splitting0 subroutine, that determines if a pair of partons can arise from the same splitting. Focusing for simplicity upon the final state radiation case, by inspecting the code we can see that a pair of coloured partons with opposite flavour, or a pair of gluons, is identified as a possible collinear pair coming from gluon splitting, and so is a flavoured coloured parton with a gluon. The electromagnetic addition involves taking into consideration an electrically charged particle together with a photon. Notice that collinear singularities arising from photon splitting into charged particle-antiparticle pairs are not considered, since it is not clear at this stage whether they would be useful or not. However, an incoming photon splitting into charged partons would be identified. Notice that the singular regions identified in this way are really emitted only if the underlying Born configuration is effectively present. Thus, unless one admits processes with incoming photon at the Born level, initial state photon splitting is not considered. In fact, at this stage, initial state photon splitting is not implemented in the rest of the `V2` code, since its usefulness will depend upon the future availability of pdf's including photons, and eventually leptons.

7.2 Computation of the NLO cross section

The implementation of electromagnetic corrections needed for the computation of the NLO cross section and the generation of the underlying Born configurations according to the $\tilde{B}$ function involves essentially three `POWHEG BOX` subsystems: the computation of the soft and collinear counterterms (in the `sigcollsoft.f` file), the computation of the softvirtual corrections (in the `sigsoftvirt.f` file), and the computation of the collinear remnants (in the `sigcollremn.f` file).

There is however a "feature" of the `POWHEG BOX` that has to be kept in mind if electromagnetic corrections are to be included. In fact, it is always assumed in the `POWHEG BOX` the real and virtual cross section supplied by the user should not carry the coupling factor $\alpha_S/(2\pi)$, that is instead supplied by the `POWHEG BOX` main subroutines. This "feature" is maintained in `V2`, so that, in practice, user processes routines for the real and virtual contributions should always be divided by $\alpha_S/(2\pi)$.

Collinear and soft subtractions should now be aware of collinear and soft singularities due to photon emission. Inspecting, for example, the `collisralr` subroutine in the `sigcollsoft.f` file, we see that, besides the standard strong interaction splitting possibilities, also the splitting of an incoming photon into a pair of opposite flavoured partons is considered, as well as photon emission from an initial state flavoured parton, making use of the Altarelli-Parisi splitting functions for electrodynamics. Notice that, in case of electromagnetic emission, the supplied coupling is the electrodynamic one, rather than the strong one. The `softalr` subroutine considers now also the possibility that the radiated parton is a photon, and computes the appropriate eikonal formula.

Softvirtual corrections should now also account for the subtracted soft and collinear electromagnetic terms. This is done by properly extending the `btildevirt` subroutine in the `sigsoftvirt.f` file to include soft and collinear corrections from electromagnetic radiation. However, the inclusion of these terms is controlled by the flag `flg_with_em`. Soft and collinear electromagnetic corrections are included here only if this flag is set to true. The reason for doing so is that it may be not always desirable to include electromagnetic virtual corrections, on the ground that NLO QED corrections are much smaller than the strong one. Yet, one may like to generate the hardest electromagnetic radiation using the `POWHEG BOX` algorithm. Not including virtual electromagnetic corrections while including instead soft-collinear one may not be fully consistent, since cancellations occour between the two kind of corrections.

7.3 Generation of radiation

The only case of generation of electromagnetic radiation that is handled by `V2` is radiation of a photon from a charged fermion. In this framework, if radiation of photons is handled in conjunction with gluon radiation, nothing special needs to be modified in the `POWHEG` code. Radiation kinematics is computed in `POWHEG` with a probability determined by the sum of all radiation processes allowed by a given underlying Born configuration. Thus, electromagnetic radiation will simply give a small correction to the distribution of the radiation variables. After the kinematics for radiation have been determined, the `POWHEG BOX` chooses one particular radiation subprocess, with a probability proportional to its cross section. Thus, photon events will be generated with the correct frequency.

When considering radiation from leptons, the photon radiation process is no longer competing with gluon radiation ones, and the subroutines for the generation of radiation must be corrected appropriately. In `V2`, in case of radiation from a colourless, but electrically charged emitter, the flag `flg_em_rad` is set true. When this flag is set, all the component of the program that deal with the generation of radiation are made aware of the fact that we are considering photon radiation. For example, the subroutine `pwhg_upperb_rad`, that computes an upper bound of the R/B function, if `flg_em_rad` is true, supplies its result multiplied by a factor of $\alpha_{\rm em}$, rather than $\alpha_{\rm S}$. Furthermore, in the `gen_rad_fsr` subroutine, the `iupperfsr=1` option is not allowed, since it assumed a leading log running coupling for radiation. In options `iupperfsr=2,3` the generation of radiation is performed with a fixed coupling constant equal to the Thomson electromagnetic coupling, rather than the strong coupling. More details on the implementation of radiation in the case of electromagnetic emission is given in Appendix B.

8. The reweighting feature

In `V2` we have introduced the reweighting feature. Its original purpose was to provide a simple method to avoid the need of a large amount of computer time and storage to perform scale and pdf variation studies. In essence, the reweighting feature allows one to generate a single event file, and then reprocess it adding new weights, corresponding to variations in the parameters that control the cross section, such as the factorization and normalization scales, the parton density functions, etc. In order to explain what the reweighting actually does, we refer the reader to the basic `POWHEG` formula for the generation of events, reported in Appendix A. The event generation in `POWHEG` takes place by first choosing whether a $\bar{B}$ or a $R_{\rm F}$ event is generated. In case of a $\bar{B}$ event, the underlying Born kinematic is generated with a probability proportional to $\bar{B}^{f_b}(\mathbf{\Phi}_n)$, while in case of an $R_{\rm F}$ event, a full real flavour structure and phase space point is generated with a probability proportional to $R_{\rm F}^{(f_r)}(\mathbf{\Phi}_{n+1})$. The reweighting feature allows to recompute either the $\bar{B}$ or the $R_{\rm F}$ weight with a different parameter setting, and to associate the new weight to the event. Notice that the generation of radiation is not altered by the reweighting procedure. On the other hand, radiation depends upon the R/B ratio, where all but one coupling constant cancel. The only remaining coupling is a power of $\alpha_{\rm S}$ that is evaluated at the transverse momentum $k_{\rm T}$. Also the parton densities cancel to some extent, although not in a complete way. In summary, we can use the reweighting feature every time we believe that the change of parameters we are considering does not affect, or affects in a negligible way, the $R_{\rm S}/B$ ratio.

We remark that although reweighting requires recalculating the amplitude, it is in fact a much faster procedure (20 to 50 times faster) than regenerating the events from scratch. In fact, no grid preparation is needed, and for each event the amplitude is calculated just once, while normally event generation requires several calls to the amplitude in order to implement the hit and miss procedure. We also remark that, in order for the reweighting to work, the importance sampling grids should be available in the directory where one performs the run. Furthermore, the settings of the `foldcsi`, `foldy` and `foldphi` variables should be the same as in the initial run.

As it will become clear in the following, a further restriction to the use of the reweighting procedure is that the Born phase space should not depend upon the parameter being varied, like for example the mass of an on-shell particle. The mass of decaying resonances may be varied, provided the phase space generator does not make use of such parameter for importance sampling purposes. In this last case, if one wishes to use the reweighting feature to perform a scan over the mass of a resonance, it is enough to separate in the Born phase space generator the mass used as an importance sampling parameter from the physical mass, and to make sure that such parameter is kept constant.

8.1 How to use the reweighting feature

In order to use the reweighting feature do the following.

- Perform a `POWHEG` run with a central parameter settings of your choice. The variable `storeinfo_rwgt` should be set to 1 in the `powheg.input` file. This is the default, so that, in fact, unless `storeinfo_rwgt` is explicitly set to zero, reweighting information will be stored after each event. At the end of the run, you will see that after each event in the Les Houches event file, a line beginning with `#rwg`, followed by some information needed for reweighting.

- To obtain new weights, corresponding to different pdf's or scale choice, or any other parameter variation allowed by the restrictions mentioned earlier, insert the line `compute_rwgt 1` in the `powheg.input` file, chage the parameters that you wish to change (typically in the `powheg.input` file, but eventually also in the code) and run the program again. A new event file will be generated, named `<OriginalName>-rwgt.lhe`, with one more line at the end of each event, of the form

  ```
  #new weight,renfact,facfact,pdf1,pdf2
  ```

  followed by five numbers and a three character string equal to either `mlm` or `lha`, yielding the new weight, the new value of the renormalization and factorization scale factors, and the pdf number for the two incoming hadrons. The `mlm` or `lha` string records whether the native pdf package was used, as compared to the lhapdf package. We remark that the really new information here is the weight of the event. All the other parameters are reported assuming that the most common use of reweighting is to vary the factorization and renormalization scales, and to try different pdf's. Other changes in the `powheg.input` file are not reported there.

One can keep modyfying the `powheg.input` and run the program again using the new event file as input, so that several `#new weight` lines are added to each event. Typically, one sets up a script that run the program several times, modifying the `powheg.input` file as needed, and renaming the output file `<OriginalName>-rwgt.lhe` back to `<OriginalName>.lhe` at the end of each run.

- It is possible to include the weight information in the format agreed upon in the 2013 Les Houche Workshop, as reported at the url `http://phystev.in2p3.fr/wiki/2013:groups:tools_lheextension`. In order to do so, add to the `powheg.input` file the lines

  ```
  lhrwgt_id 'xxx'
  lhrwgt_descr 'some info'
  lhrwgt_group_name 'some name'
  lhrwgt_group_combine 'foo'
  ```

  The produced `.lhe` file will carry in the header the following lines

  ```
  <header>
  <initrwgt>
  <weightgroup name='some name' combine='foo' >
  <weight id='xxx'> some info </weight>
  </weightgroup>
  </initrwgt>
  </header>
  ```

  If you omit `lhrwgt_group_name`, no weightgroup lines will appear. If you omit the `lhrwgt_group_combine` line, no `combine='foo'` will appear. If you omit the `lhrwgt_descr` line, the `'some info'` string will be taken as empty. It is mandatory that at least the `lhrwgt_id` line be present in the `powheg.input` file in the initial run. In the reweighting runs, inserting the lines

  ```
  lhrwgt_id 'yyy'
  lhrwgt_descr 'some info'
  lhrwgt_group_name 'some name'
  lhrwgt_group_combine 'foo'
  ```

  will cause the Les Houches reweight information to be updated. If the group name is new, a new group is added to the header. If it is already in the header, it will not be modified (i.e. the `lhrwgt_group_combine` line will be ignored), and the id yyy will be inserted in the existing group. If no group is mentioned, the weight will be added out of any `weightgroup` tag. Furthermore, at the end of each event the initial and new weight line will be written

```
<rwgt>
<wgt id='xxx'>  weight </wgt>
<wgt id='yyy'>  weight </wgt>
</rwgt>
```

Notice that `lhrwgt_id` can be used only if the initial run, with the `storeinfo_rwgt` line, also carries a `lhrwgt_id`. Otherwise the `initrwgt` header is not found and the program aborts. Also, if the id name supplied in the `powheg.input` file is already present in the `.lhe` file, the program aborts with a warning.

8.2 Implementation of the reweighting feature

The `#rwg` line contains the following information:

- An integer from 1 to 3, corresponding to the `rad_type` value for the current event, where 1 corresponds to a $\bar{B}$ event, and 2 and 3 correspond both to $R_{\rm F}$ events. In `POWHEG` these are internally separated in terms that have a one to one correpoondence to radiation regions, called damp remnants, that have `rad_type 2`, and one that corresponds to real contributions that have no singular regions, the so called regular remnants, that have `rad_type` equal to 3.

- An index that labels which (flavour structure) contribution is being currently generated. This is stored in the `rad_ubornidx` variable if `rad_type=1`, `rad_realalr` if `rad_type=2`, and `rad_realreg` if `rad_type=2`

- The value of the corresponding squared amplitude, that is usually stored in the `rad_btilde_arr(rad_ubornidx)` in the first case, `rad_damp_rem_arr(rad_realalr)` in the second case, and `rad_reg_arr(rad_realreg)` in the third case.

- The value of the 3 random seeds right before the relevant amplitude square is calles. Thanks to these values, ohne will be able to recalculate the same phase point during the reweighting phase.

These values are written at the end of each event by the `lhefwriteevrw` subroutine, in the `lhefwrite.f` file, provided the `flg_reweight` flag is set to true, which in turns is the case unless `storeinfo_rwgt` is set to zero in the `powheg.input` file. The event generation mechanism in `POWHEG` guarantees that at the time when the Les Houches event is written all the other reweight information parameters have the settings appropriate for the current event. As far as the random seen is concerned, the `gen` subroutine (in the `integrator.f` file), that is responsible for the generation of the underlying Born and remnants kinematics, was modified, so that before each hit and miss trial the status of the event generator (i.e.

the IJKL, NTOT and NTOT2 parameters in the RM48 CERNLIB subroutines) is stored in three common block variables.[3] This is done by the readcurrentrandom call in gen.

The heart of the reweighting mechanism is the pwhgnewweight(iunin,iunrwgt) subroutine (in the pwhgreweight.f file), that is invoked in the POWHEG main program instead of the event generation routine if the flg_newweight flag is set to true. It takes as arguments the unit numbers of the input and output .lhe files (i.e. the one that is red, and the one that is written with the new weight information). It first reads in the event, copying it to the output file, and stores the reweight information. This is done by the subroutine lhefreadevnew, that is defined in the same file. It sets the stored value of the squared amplitude in the rad_currentw variable, it initializes the random number seeds to the values red in the input file, and, depending upon rad_type, calls the gen_btilderw o the gen_sigremnantrw subroutines. These subroutines call the gen routine, passing as first argument either the btilde or the sigremnant routine, setting the imode input parameter to 2. When this is the case, the gen subroutine just performs a single call to the function, rather than following the standard hit and miss procedure, ending up with the arrays rad_btilde_arr, rad_damp_rem_arr or rad_reg_arr (whichever applies) being filled with the recomputed value. The new weight is then obtained by multiplying the weight of the Les Houches event by the ratio of the newly computed matrix element over the rad_currentw value.

A. Basic POWHEG formulae

In this appendix we give for convenience the basic POWHEG formulae that are implemented in the POWHEG BOX package. We denote by f_b the Born flavour structures that contribute to our cross section. These are represented by the flst_born(1:nlegborn,1:flst_nborn) array in the POWHEG BOX. In POWHEG the real cross section is partitioned as follows

$$R = R_{\rm S} + R_{\rm F}\,, \qquad R_{\rm S} = \sum_{\alpha_{\rm r}} R_{\rm S}^{(\alpha_{\rm r})}\,, \tag{A.1}$$

where $R_{\rm F}$ (if present) does not have any collineas or soft singularities, and $R_{\rm S}$ is separated into components labelled by the $\alpha_{\rm r}$ index, that separate the real cross section into different real flavour structure, and within each flavour structure also separate contributions that can have collinear and soft singularities only in a specific phase space region. The formula

[3]The CERNLIB distributed version of the RM48 has a bug, that was spotted and fixed during the construction of the reweighting feature. The call to RM48IN(ISEED,N1,N2), that should bring the generator in a state of initialization with seed ISEED after N1+N2*10**9 calls, if N2>0, fails. The version of RM48 included in the cernroutines.f file in POWHEG has been fixed for this bug since svn revision 2159.

for the full POWHEG cross section is

$$
\begin{aligned}
d\sigma = &\sum_{f_b} \bar{B}^{f_b}(\mathbf{\Phi}_n)\, d\mathbf{\Phi}_n \left\{ \Delta^{f_b}\left(\mathbf{\Phi}_n, p_{\rm T}^{\min}\right) \right. \\
&+ \left. \sum_{\alpha_{\rm r}\in\{\alpha_{\rm r}|f_b\}} \frac{\left[d\Phi_{\rm rad}\, \theta\left(k_{\rm T} - p_{\rm T}^{\min}\right) \Delta^{f_b}(\mathbf{\Phi}_n, k_{\rm T})\; R_{\rm S}\left(\mathbf{\Phi}_{n+1}\right)\right]_{\alpha_{\rm r}}^{\bar{\mathbf{\Phi}}_n^{\alpha_{\rm r}}=\mathbf{\Phi}_n}}{B^{f_b}(\mathbf{\Phi}_n)} \right\} \\
&+ \sum_{f_r} R_{\rm F}^{(f_r)}(\mathbf{\Phi}_{n+1})\; d\mathbf{\Phi}_{n+1}\,. \qquad \text{(A.2)}
\end{aligned}
$$

with the POWHEG Sudakov form factor given by

$$
\Delta^{f_b}(\mathbf{\Phi}_n, p_{\rm T}) = \exp\left\{ - \sum_{\alpha_{\rm r}\in\{\alpha_{\rm r}|f_b\}} \int \frac{\left[d\Phi_{\rm rad}\, R_{\rm S}\left(\mathbf{\Phi}_{n+1}\right)\, \theta\left(k_{\rm T}(\mathbf{\Phi}_{n+1}) - p_{\rm T}\right)\right]_{\alpha_{\rm r}}^{\bar{\mathbf{\Phi}}_n^{\alpha_{\rm r}}=\mathbf{\Phi}_n}}{B^{f_b}\left(\mathbf{\Phi}_n\right)} \right\}. \qquad \text{(A.3)}
$$

With f_b we denote each flavour structure that contribute at the Born level, and the $\alpha_{\rm r}$ index labels each singular component of the real cross section. The square bracket $[]_{\alpha_{\rm r}}$ means that the real contribution R, the transverse momentum $k_{\rm T}$, and the partition of phase space $\mathbf{\Phi}_{n+1} = \mathbf{\Phi}_n \times \Phi_{\rm rad}$ all refer to the $\alpha_{\rm r}$ singular region. The notation $\alpha_{\rm r} \in \{\alpha_{\rm r}|f_b\}$ means: all singular regions that have f_b as underlying Born configuration.

For ease of notation, we have dropped the $\mathbf{\Phi}_{n+1}$ argument in $k_{\rm T}^{\alpha_{\rm r}}$. The $p_{\rm T}^{\min}$ value introduced here is a lower cut-off on the transverse momentum, that is needed in order to avoid to reach unphysical values of the strong coupling constant and of the parton-density functions.

The generation of an event in POWHEG begins by first choosing if it comes from the first or second term of eq. (A.2), with a probability proportional to the contribution they give to the total cross section. If it comes from the first term, a born flavour f_b and a Born kinematics *Kinn* is chosen with a probability proportional to $\bar{B}^{f_b}(\mathbf{\Phi}_n)$. This kinematic configuration gives rise to events with a Born kinematics, with a probability equal to the first term in the curly braket, or with further radiation, with kinematics specified by a mapping from the $\mathbf{\Phi}_n$ and $\Phi_{\rm rad}$ variables into the *Kinnpo* phase space, with a probability equal to each term in the sum. Notice that the sum of all terms in the curly bracket yields one, so that adding the radiation to the event does not change the initial probability, as is typically the case in shower algorithms.

In case the second term of eq.. (A.2) is chosen, a real flavour configuration f_r and a real kinematics $\mathbf{\Phi}_{n+1}$ is chosen with a probability proportional to $R_{\rm F}^{(f_r)}(\mathbf{\Phi}_{n+1})$.

B. Generation of final state radiation

The purpose of this appendix is to integrate and clarify what was exposed in appendix C in ref. [5].

First of all, we remind the reader that the initial generation of radiation is performed with an approximate form of the R/B ratio, such that the generation of the transverse momentum of radiation can be performed in a reasonably simple way. First of all, we need a function

$$U_0(\xi, y) = N\, u_0(\xi, y), \tag{B.1}$$

such that the expression

$$\Delta^{(U)}(p_{\rm T}) = \exp\left[-\int U(\xi, y)\, \theta(k_{\rm T} - p_{\rm T})\; d\xi\, dy\, d\phi \right]. \tag{B.2}$$

can be computed analytically for any value of N. A second function is also introduced

$$U(\xi, y) = N u(\xi, y), \tag{B.3}$$

with $u(\xi, y) < u_0(\xi, y)$. The constant normalization N is computed in such a way that $U(\xi, y) \geq \sum R/B$, where $\sum R/B$ is intended to run over all possible radiations at a given underlying Born flavour. Thus

$$N = \max \frac{\sum R/B}{U(\xi, y)}. \tag{B.4}$$

The function $u(\xi, y)$ is coded in the `pwhg_upperb_rad()` function. By inspection, one can see that this function handles both initial and final state radiation, and that the form of the function can be chosen among a few cases according to the value of the `rad_iupperisr` and `rad_iupperfsr` flags (that in turn are set according to the variables `iupperisr` and `iupperfsr` in the `powheg.input` file). Furthermore, the case of emission from a massive emitter is handled as a special case. We remark that in all cases the $u(\xi, y)$ carry a factor of the two-loop, `POWHEG BOX` running coupling constant, `st_alpha`, except in the case of electromagnetic emission, that carries the Thomson value of the electromagnetic coupling.

The form of the $u(\xi, y)$ function for ISR, `iupperisr 1`, is

$$u(\xi, y) = \frac{\alpha_{\rm S}}{\xi(1 - y^2)}\,, \tag{B.5}$$

with no other options offered at the moment. In the FSR case, for `iupperfsr 1` we have

$$u(\xi, y) = \frac{\alpha_{\rm S}}{\xi(1 - y)}\,, \tag{B.6}$$

for `iupperfsr 2` (the default value) we have

$$u(\xi, y) = \frac{\alpha_{\rm S}}{(\xi^2(1 - y)(1 - \xi/2(1 - y))^2)}\,, \tag{B.7}$$

and for `iupperfsr 3` we have

$$u(\xi, y) = \frac{\alpha_{\rm S}}{(\xi(1 - y)(1 - \xi/2(1 - y)))}\,. \tag{B.8}$$

The case of massive emitter is handled as a special case according to ref. [10]. The corresponding u_0 functions are given as follows. for ISR, `iupperisr 1`, we have

$$u_0(\xi, y) = \frac{\alpha_{\rm S}^{(0)}}{\xi(1 - y^2)}\,, \tag{B.9}$$

and similarly, for FSR, `iupperfsr 1`

$$u_0(\xi, y) = \frac{\alpha_S^{(0)}}{\xi(1-y)}\,, \tag{B.10}$$

where $\alpha_S^{(0)}$ is the one loop QCD running coupling with 5 flavours:

$$\alpha_S^{(0)} = \frac{1}{\frac{33-10}{12\pi}\log\frac{p_T^2}{\Lambda^2}}. \tag{B.11}$$

In the FSR case, for both `iupperfsr` equal to 2 and 3, we use

$$u_0(\xi, y) = \frac{1}{(\xi^2(1-y)(1-\xi/2(1-y))^2)}. \tag{B.12}$$

The generation of the radiation scale starts initially with the U_0 distributions, computed analytically according to appendix C in ref. [5]. In order to go from the u_0 to the u upper bound, further vetoing is needed. In case of ISR and FSR radiation with `iupperisr` or `iupperfsr` equal to 1, one needs to go from $\alpha_S^{(0)}$ to the two loop α_S. This is achieve by vetoing the radiation with a probability proportional to $1-\alpha_S/\alpha_S^{(0)}$. On the other hand, in the `iupperfsr` equal to 2 or 3 cases, one vetoes radiation with a probability proportional to $1-\alpha_S$, in order to supply the missing α_S factor. Similarly, the procedure for implementing the `iupperfsr=3` case is the same as for `iupperfsr=2`, except that an extra factor

$$\frac{u_0^{(3)}}{u_0^{(2)}} = \xi(1-\xi/2(1-y)) = \xi - \frac{t}{s}\,, \tag{B.13}$$

(where the upper index refers to the `iupperfsr` value) is supplied by vetoing radiation with a probability proportional to $1-u_0^{(3)}/u_0^{(2)}$ (we have used the fact that the transverse momentum for radiation is defined to be $t = s\xi^2(1-y)/2$, see the `pwhg_pt2` function).

In case of FSR electromagnetic radiation, `iupperfsr=1` cannot be implemented, while the values 2 and 3 can be directly implemented including the electromagnetic coupling in the u_0 function, without need of further vetoing afterwards. The electromagnetic coupling factor is supplied directly at the beginning of the `pt2solve` function, and it is explicitly removed from the `gen_rad_fsr` subroutine, yielding a typical example of questionable coding practices.

References

[1] K. Hamilton, P. Nason, and G. Zanderighi, *MINLO: Multi-Scale Improved NLO*, *JHEP* **1210** (2012) 155, [1206.3572].

[2] K. Hamilton, P. Nason, C. Oleari, and G. Zanderighi, *Merging H/W/Z + 0 and 1 jet at NLO with no merging scale: a path to parton shower + NNLO matching*, *JHEP* **1305** (2013) 082, [1212.4504].

[3] S. Catani, F. Krauss, R. Kuhn, and B. Webber, *QCD matrix elements + parton showers*, *JHEP* **0111** (2001) 063, [hep-ph/0109231].

[4] J. M. Campbell, R. K. Ellis, P. Nason, and G. Zanderighi, *W and Z bosons in association with two jets using the POWHEG method*, *JHEP* **1308** (2013) 005, [1303.5447].

[5] S. Alioli, P. Nason, C. Oleari, and E. Re, *A general framework for implementing NLO calculations in shower Monte Carlo programs: the POWHEG BOX*, *JHEP* **1006** (2010) 043, [1002.2581].

[6] S. Frixione, E. Laenen, P. Motylinski, and B. R. Webber, *Angular correlations of lepton pairs from vector boson and top quark decays in Monte Carlo simulations*, *JHEP* **0704** (2007) 081, [hep-ph/0702198].

[7] S. Frixione, P. Nason, and C. Oleari, *Matching NLO QCD computations with Parton Shower simulations: the POWHEG method*, *JHEP* **0711** (2007) 070, [0709.2092].

[8] P. Nason, *MINT: A Computer program for adaptive Monte Carlo integration and generation of unweighted distributions*, 0709.2085.

[9] T. Melia, P. Nason, R. Rontsch, and G. Zanderighi, *W^+W^+ plus dijet production in the POWHEGBOX*, *Eur.Phys.J.* **C71** (2011) 1670, [1102.4846].

[10] L. Barze, G. Montagna, P. Nason, O. Nicrosini, and F. Piccinini, *Implementation of electroweak corrections in the POWHEG BOX: single W production*, *JHEP* **1204** (2012) 037, [1202.0465].