12 Loop calculations: numerical approach

The computation of Feynman loop diagrams has a long history in High-Energy physics. The analytical treatment of the triangle loop integral of the previous exercise offered a glimpse at some aspects this technology, such as the Feynman parameterisation, d-dimensional regularisation and tensor reduction. The current state-of-the-art in loop calculations is full automation at one loop, and dedicated methods beyond one-loop that smartly combine analytical and numerical techniques.

In this exercise, you will explore a completely different and fully numerical alternative for the computation of the same triangle integral. You will implement the numerical solution in a `Python+Rust` computer code, starting from a template implementation provided to guide your work.

For the toy model of the previous exercise, we had found that the scalar decay width Γ stemming from the process $S \to \gamma\gamma$ reads:

$$\Gamma(S \to \gamma\gamma) = \frac{1}{16\pi}\frac{1}{m_S}\sum_{\lambda,\lambda'} \left|\mathcal{M}_{\lambda,\lambda'}\right|^2, \tag{1}$$

thanks to the fact that the $1 \to 2$ phase-space integral can be performed analytically, and it factorises entirely the squared matrix element $\mathcal{M}_{\lambda,\lambda'}$. Now, assigning the four-momentum k to the edge of the triangle between the two photons, each carrying incoming momentum q and p respectively, we found that that the helicity amplitude $\mathcal{M}_{\lambda,\lambda'}$ could be written:

$$\begin{aligned}
\mathcal{M}_{\lambda,\lambda'}(q,p) &= ie^2Q^2\frac{m_\psi}{v}\epsilon_\mu(p,\lambda)\epsilon_\nu(q,\lambda')I^{\mu\nu}(q,p) && (2)\\
I^{\mu\nu}(q,p) &= \int \frac{d^4k}{(2\pi)^4}\frac{N^{\mu\nu}(k,q,p)}{(k^2-m_\psi^2+i\epsilon)((k-q)^2-m_\psi^2+i\epsilon)((k+p)^2-m_\psi^2+i\epsilon)} && (3)\\
N^{\mu\nu}(k,q,p) &= \mathrm{Tr}\left[(\not{k}+m_\psi)\gamma^\mu(\not{k}+\not{p}+m_\psi)(\not{k}-\not{q}+m_\psi)\gamma^\nu\right] + (p\leftrightarrow q \;\&\; \mu\leftrightarrow\nu)\\
&= 8m_\psi[4k^\mu k^\nu - g^{\mu\nu}k^2 - p^\mu q^\nu + p^\nu q^\mu - g^{\mu\nu}\frac{m_S^2}{2} + g^{\mu\nu}m_\psi^2]. && (4)
\end{aligned}$$

This is the point at which we had introduced the Feynman parameterisation, and proceeded with tensor reduction of the numerator and application of d-dimensional integral identities.

12.1 Theory part

We will now take a different route, and compute the integral $I^{\mu\nu}(q,p)$ numerically directly in momentum space. To this end, a first concern may be the $k^\mu k^\nu$ and k^2 terms in the numerator.

1.1a Let us define $T^{\mu\nu}$ the tensor integral corresponding to the integral of Eq. (3) with $N^{\mu\nu} = k^\mu k^\nu$. Argue that Lorentz invariance allows one to derive the following decomposition (with unspecified coefficients α_i):

$$T^{\mu\nu} = \alpha_1\, g^{\mu\nu} + \alpha_2\, p^\mu q^\nu + \alpha_3\, q^\mu p^\nu + \alpha_4\, p^\mu p^\nu + \alpha_5\, q^\mu q^\nu. \tag{5}$$

1.1b [OPTIONAL] In a previous exercise, we had derived the spin-sum relation for a *massive* vector. In the case of a massless vector, one can show that the spin-sum rule can be written as:

$$\sum_{\lambda,\lambda'\in\{-1,1\}} \epsilon_\lambda^\mu(p)\epsilon_{\lambda'}^{\nu\star}(p) = -g^{\mu\nu} + \frac{n^\mu p^\nu + p^\mu n^\nu}{p\cdot n} - \frac{p^\mu p^\nu}{(p\cdot n)^2}, \tag{6}$$

for an arbitrary time-like normalized vector satisfying $n^\mu n_\mu = 1$, identifying the axial gauge choice for the photon polarisation vectors, with the property of $\epsilon^\mu n_\mu = 0$. Note that the four-vector n can in principle be chosen independently for each photon, and a smart choice can help simplify the computation. In particular, choosing $n = (1, 0, 0, 0)$ and defining $\bar{p}^\mu \equiv 2(p\cdot n)n^\mu - p^\mu$ (meaning $\bar{p}^\mu = (p^0, -\vec{p})$, with $p^\mu = (p^0, \vec{p})$), show that one obtains:

$$\sum_{\lambda,\lambda' \in \{-1,1\}} \epsilon^\mu_\lambda(p)\epsilon^{\nu\star}_{\lambda'}(p) = -g^{\mu\nu} + \frac{p^\mu \bar{p}^\nu + \bar{p}^\mu p^\nu}{p\cdot\bar{p}}. \tag{7}$$

Assuming the above, show that in the rest frame of the decaying scalar S (where the two photons are back-to-back, so that $q = \bar{p}$), terms of $I^{\mu\nu}$ proportional to $p^\mu q^\nu$, $q^\mu p^\nu$, $p^\mu p^\nu$ and $q^\mu q^\nu$ do not contribute to the unpolarized decay rate of Eq. (1).
Hint: For physical on-shell photons, remember the orthogonality condition $\epsilon_\lambda(p)\cdot p = 0$. Then, also use the axial gauge property $\epsilon_\lambda(p)\cdot n = 0$ in the expansion of $\epsilon_\lambda(p)\cdot q$ with $q = \bar{p}$.

1.1c Argue that the first two numerator terms $4k^\mu k^\nu$ and $g^{\mu\nu}k^2$ would *locally* induce an Ultra-Violet (UV) divergence when the norm of the 4-momentun k^μ approaches infinity during the numerical integration of Eq. (3). However, thanks to the result from Ex. 1.1b, we now know that we can choose to perform our computation in a particular gauge in which we can ignore numerator terms proportional to $p^\mu q^\nu$ and $q^\mu p^\nu$ and simply write $T^{\mu\nu} = \alpha g^{\mu\nu}$.

Show then that $\alpha = \frac{1}{4}g^{\mu\nu}T^{\mu\nu}$, so that we can rewrite the original integral $I^{\mu\nu}(q,p)$ of Eq. (3) with $N^{\mu\nu} = g^{\mu\nu}(m_\psi^2 - m_S^2/2)$. This much simpler numerator is now independent of the loop momentum k^μ and free of UV divergences. However, remember that this modified numerator will only yield the same result as the original one when working in the rest frame of S and for our particular axial gauge choice.

Thanks to the above realisation, we find that the tensor numerator is now independent of the loop momentum and can be taken outside the loop integral (this does not happen in general of course, but it thankfully simplifies our life here). Provided that we work in the rest frame of the decaying particle S and that the polarization vectors ϵ^μ are expressed in the axial gauge with $n = (1, 0, 0, 0)$, the helicity amplitude of Eq. (2) can be rewritten as:

$$\mathcal{M}_{\lambda,\lambda'}(q,p) = 8e^2Q^2\frac{m_\psi^2}{v}\epsilon_\mu(p,\lambda)\epsilon_\nu(q,\lambda')g^{\mu\nu}(m_\psi^2 - m_S^2/2)I(q,p) \tag{8}$$

$$I(q,p,m_\psi) = \int d^4k \frac{i}{(2\pi)^4}\frac{1}{(k^2 - m_\psi^2 + i\epsilon)((k-q)^2 - m_\psi^2 + i\epsilon)((k+p)^2 - m_\psi^2 + i\epsilon)}. \tag{9}$$

1.2 Compute analytically the scalar integral $I(q,p,m_\psi)$, for $m_S < 2m_\psi$, using a derivation very close to what you did in the previous exercise. You should find:

$$I(q,p) \equiv I(2q\cdot p = m_S^2, m_\psi) = \frac{1}{8\pi^2}\frac{1}{m_S^2}\arcsin^2\left[\frac{m_S}{2m_\psi}\right]. \tag{10}$$

Which symmetry guarantees that the result only depends on the internal propagator mass and the quantity $q\cdot p$? Quickly check the dimensional consistency of this result as well.

1.3 Characterize the location in $\vec{k}$ of all singularities in the four-dimensional expression of Eq. (6). Sketch what they look like on a two-dimensional plane (k^0, k_x).

1.4 Derive the *Loop-Tree Duality* (LTD) expression for the triangle integral of Eq. (6). It is obtained by analytically performing the integral in dk^0 using Cauchy theorem and a contour integral closed using a semi-circle with infinite radius in the lower-half complex plane. Explain why the integral along this semi-circle is zero, and you should find:

$$I(q,p,m_\psi) = \int d^4k \frac{1}{(2\pi)^3}\left[\frac{\delta^+(k^2-m_\psi^2)+\delta^+((k-q)^2-m_\psi^2)+\delta^+((k+p)^2-m_\psi^2)}{(k^2-m_\psi^2+i\epsilon)((k-q)^2-m_\psi^2+i\epsilon)((k+p)^2-m_\psi^2+i\epsilon)}\right] \quad (11)$$

where $\delta^+(q^2-m_\psi^2) \equiv (q^2-m_\psi^2+i\epsilon)\theta(q^0)\delta(q^2-m_\psi^2)$. Give a diagrammatic interpretation of this expression which justifies the name of *Loop-Tree Duality*. You should obtain Eq. (11) by introducing the δ^+ as an operational way of taking the residues selected by your choice of residues (i.e. you do not need to mathematically derive it). Now solve the δ^+ distributions in order to explicitly find the following 3-dimensional integral representation:

$$\begin{aligned} I(q,p,m_\psi) = & \int d^3\vec{k}\frac{1}{(2\pi)^3}\Big[\\ & \frac{1}{2E_1}\frac{1}{(\bar\eta_{12}^{++}-q^0)(\bar\eta_{12}^{+-}-q^0)}\frac{1}{(\bar\eta_{13}^{++}+p^0)(\bar\eta_{13}^{+-}+p^0)} \\ & +\frac{1}{(\bar\eta_{21}^{++}+q^0)(\bar\eta_{21}^{+-}+q^0)}\frac{1}{2E_2}\frac{1}{(\bar\eta_{23}^{++}+p^0+q^0)(\bar\eta_{23}^{+-}+p^0+q^0)} \\ & +\frac{1}{(\bar\eta_{31}^{++}-p^0)(\bar\eta_{31}^{+-}-p^0)}\frac{1}{(\bar\eta_{32}^{++}-p^0-q^0)(\bar\eta_{32}^{+-}-p^0-q^0)}\frac{1}{2E_3}\Big], \end{aligned} \quad (12)$$

where we have defined $E_i = \sqrt{|\vec{k}+\vec{q}_i|^2+m_\psi^2}$, with $q_i = \{0,-q,p\}$, and $\bar\eta_{ij}^{\sigma_i\sigma_j} = \sigma_i E_i + \sigma_j E_j$ with $\sigma_k \in \{-1,1\}$. Notice that the expression above can be written even more concisely by introducing the following notation including energy shifts: $\eta_{ij}^{\sigma_i\sigma_j} = \sigma_i E_i + \sigma_j E_j + \sigma_i(q_j^0 - q_i^0)$:

$$\begin{aligned} I(q,p,m_\psi) = & \int d^3\vec{k}\frac{1}{(2\pi)^3}\Big[\\ & +\frac{1}{2E_1}\frac{1}{\eta_{12}^{++}\eta_{12}^{+-}}\frac{1}{\eta_{13}^{++}\eta_{13}^{+-}} + \frac{1}{\eta_{21}^{++}\eta_{21}^{+-}}\frac{1}{2E_2}\frac{1}{\eta_{23}^{++}\eta_{23}^{+-}} + \frac{1}{\eta_{31}^{++}\eta_{31}^{+-}}\frac{1}{\eta_{32}^{++}\eta_{32}^{+-}}\frac{1}{2E_3}\Big]. \end{aligned} \quad (13)$$

1.5 We refer to η_{ij}^{+-} and η_{ij}^{-+} as Hyperobilic surfaces, or H-surfaces, and η_{ij}^{++} as Ellipsoid surfaces, or E-surfaces. Notice that $\eta_{ij}^{+-} = -\eta_{ji}^{+-}$. The surface defined by the solution of $\eta_{ij}^{\pm\pm}(\vec{k}) = 0$ is important as it defines the location of potential singularities preventing direct numerical integration over the loop momentum $\vec{k}$.

1.5.a Indicate on the 2D diagram (k^0, k_x) of Ex. 1.3 where the singularities of the H- and E-surfaces are located.

1.5.b Sketch what the E-surface defined by $\eta_{23}^{++}(\vec{k}) = 0$ looks like for $m_\psi = 0$ and $p = (1, 0, 0, 1), q = (2, 0, 0, -2)$ in the (k_x, k_y) plane, and then in the (k_y, k_z) plane.

1.5.c Work out the existence condition of all three E-surfaces η_{12}^{++},η_{31}^{++} and η_{32}^{++} (when can $\eta_{ij}^{++}(\vec{k}) = 0$ have a solution for these surfaces?).

Hint: Consider a generic E-surface $\eta(\vec{k}) = \sqrt{|\vec{k}|^2 + m^2} + \sqrt{|\vec{k} - \vec{p}|^2 + m^2} + E$ *(one can always choose one square root to have no shift using a change of variable* $k^\mu \to k^\mu + q^\mu$ *with an appropriate choice of* q^μ*; note that this will also affect the quantity E). Then,* $\eta(\vec{k}) = 0$ *will have a solution iff a.* $\exists\ \vec{k}$ *s.t.* $\eta(\vec{k}) > 0$ *and b.* $\exists\ \vec{k}$ *s.t.* $\eta(\vec{k}) < 0$.
Argue that condition a. can always be met by choosing $|\vec{k}|$ *sufficiently large. Then, to find out when condition b. can be met, the triangle inequality can be used to argue (you do not have to show this) that the minimum of* $\eta(\vec{k})$ *is obtained at* $\vec{k} = \frac{1}{2}\vec{p}$*. The existence condition can then be obtained from investigating when* $\eta(\frac{1}{2}\vec{p}) < 0$.

1.5.d [OPTIONAL] Re-interpret the existence condition obtained in Ex. 1.5.c to show that $\eta_{ij}^{++} = 0$ has no solution unless $m_\psi = 0$, or $\sqrt{(p+q)^2} = \sqrt{p_S^2} = m_S > 2m_\psi$.

1.6 [OPTIONAL] The previous exercise demonstrated that the E-surface singularities are not problematic when we have either non-physical so-called *euclidean* kinematics, where $p^2 < 0$, $q^2 < 0$ and $(p+q)^2 < 0$, or when $m_S < 2m_\psi$ and $m_\psi > 0$. There however remains potential H-surface singularities. Thankfully, and somewhat amazingly, these can be shown to be spurious singularities, in the sense that they *all* cancel pair-wise between the three terms of Eq. 10 (each of these terms is typically refered to as a cut). It is even possible to make such cancellations manifest by algebraically manipulating Eq. 10. Show that it can be identically rewritten as:

$$I^{(\text{improved})}(q, p, m_\psi) = \int d^3\vec{k} \frac{1}{(2\pi)^3} \frac{1}{(2E_1)(2E_2)(2E_3)} \Big[\frac{1}{\eta_{31}^{++}\eta_{32}^{++}} + \frac{1}{\eta_{12}^{++}\eta_{32}^{++}} + \frac{1}{\eta_{12}^{++}\eta_{13}^{++}} + \frac{1}{\eta_{13}^{++}\eta_{23}^{++}} + \frac{1}{\eta_{21}^{++}\eta_{23}^{++}} + \frac{1}{\eta_{21}^{++}\eta_{31}^{++}} \Big]. \tag{14}$$

Hint: Rewrite Eq. 10 using, at opportune places, the partial fractioning identity: $\frac{1}{xy} = \frac{1}{y-x}\left(\frac{1}{x} - \frac{1}{y}\right)$
The expression is now manifestly free of H-surface singularities, and is also numerically more stable since it does not involve large cancellations between cuts. This improved LTD expression is therefore superior for both theoretical singularity analysis (all physical singularities can be read directly from the denominators), but also for numerical implementations. A very elegant algorithm was recently established for systematically deriving such improved LTD expressions (called the *Cross-Free Family* representation) for any (multi-)loop integral, see `https://arxiv.org/pdf/2211.09653.pdf`.

12.2 Code part

We are now ready to perform the numerical implementation of the integral of Eq. (10). In the code, we will not concern ourselves with explicitely relating that integral to the decay rate, as this is fairly trivial as shown above in Eqs. (1) and (2). Here are some general guidelines to this part:

a. The code template you will start from is provided to you through the tarball named `numerical_code.tar.gz`. It requires `Python3.11+`, and also `Rust` for an optional part of exercise. The necessary `Python3.11+` dependencies can be installed with `pip`. You can download the tarball from the course website and extract it with the following command:

```
tar -xzf numerical_code.tar.gz
```

The `Python` file containing the `main` code is called `triangler.py`.

b. It is recommended to write your code using the Visual Studio Code (VSC) Integrated Development Environment (IDE). If you are working with the computers in rooms A94 or A95 of the University of Bern, then you can immediately get VSC and `Python3.11+` by downloading the tarball `UBUNTU_RESOURCES.tar.gz` containing pre-compiled binaries.

c. As some parts of this assigment are more advanced, they are indicated as optional. Their completion can however increase the grade of this exercise by up to a bonus of 40% (with the possibly of exceeding 100%). The optional final challenge can grant an additional 10% bonus.

d. This exercise sheet does *not* contain all the technical details necessary to complete the code. Similarly as when carrying an actual research project, you will need to consult relevant online resources and/or pro-actively ask tutors for help.

e. All parts of the code raising a `NotImplemented Python` error with a message pointing to an exercise number indicates parts of the code you are supposed to fill in.

f. You are expected to submit the completed code as a tarball named `numerical_code_solution_<your_name>.tar.gz`, including within it a report (named `numerical_code_report.pdf`) describing your solution for the theoretical exercises as well as for the code exercises when explanations are requested.
Also include a file `commands.makefile` that lists the commands (and only those) you use to test the implementation of *each* coding exercise you completed. Example content of this file:

```
all: Ex2.1 Ex2.2 # List here all exercises you completed
Ex2.1:
    echo "Commands for Ex2.1"
    python3 triangler.py ... # First command
    python3 triangler.py ... # Second command, etc..
Ex2.2: # Place your commands below for Ex2.2, and similarly for all others
```

The advantage is then that you can easily run all your commands with `make -f commands.makefile`, or those for one particular exercise with, e.g., `make -f commands.makefile Ex2.2`.

g. The solution code is also provided to you with the password-protected zipped folder `numerical_code_solution.zip`. The password will be given to you after submission.

h. Do not hesitate to ask technical questions on our Zulip collaborative chat in order to allow you to quickly progress with the code.

2.1 You can obtain help for the available commands of the code `triangler.py` by running:

```
python3 triangler.py --help; python3 triangler.py <subcommand> --help
```

However, the template code you receive will not run the above yet, as you first exercise is to add the `analytical_result` subcommand in the command line parser, as well as the `--m_psi` option (with default value 0.02) to allow the user to specify the mass of the internal propagator. You must then implement the function `analytical_result` which specifies our target analytical result of Eq. (7). You should now be able to obtain the following output:

```
python3 triangler.py analytical_result
{...} Analytical result: +8.0863564465181099e+00 +0j GeV^{-2}
```

This will be our target result for the numerical evaluation of the integral of Eq. 6.

2.2 Implement the 3-dimensional LTD integrand $I(q,p)$ of Eq. (10) in the `Python` function `python_integrand` provided in the template code. You can run the `inspect` subcommand to test your implementation:

```
python3 triangler.py inspect --point 0.1 0.2 0.3
{...} Integrand evaluated at loop momentum k = {...} : +1.0608105032736712e-01
```

Notice that you can add additional debug printouts to help debug your implementation with `logger.debug("...")` statements that will only be shown on the screen when running the code with the `-v debug` option.

2.3 [OPTIONAL] Implement the improved version of the LTD expression of Eq. (11) in the same `python_integrand` function. You can test your implementation with the `inspect` subcommand again, but with the main option `--improved_ltd`. You should find the same result as before, but with progressively more different trailing digits as your increase the norm of the test point $\vec{k}$ (because of numerical instabilities plaguing the naive implementation).

2.4 [OPTIONAL] `Python` offers a lot of flexibility but not much performance. Other compiled languages, like `Rust` can offer much better run-time performances at the expense of a more complex syntax. It is however possible to combine the best of both worlds with hybrid implementations. To illustrate this, implement the improved LTD expression in the `rust_integrand` function and the `src/lib.rs` (which uses `https://pyo3.rs/v0.20.0` to allow `Python` to call `Rust` functions). You can again test your implementation with the options `--improved_ltd -ii rust`.

2.5 We must now parameterise the integration space. Implement both cartesian and spherical coordinate parameterisations of the $\mathbb{R}^3$ domain from the unit hypercube $(0,1)^3$ in the functions `cartesian_parameterize` and `spherical_parameterize`. A map from the compact interval $(0,1)$ to the infinite domain $(-\infty,\infty)$ is called a conformal map. The polynomial map $r(x) = 1/(1-x) - 1/x$ is such a map. For the spherical parameterisation, use an anologuous conformal polynomial map mapping to the domain $(0,\infty)$ instead. For the cartesian parameterisation, use a similar conformal map using logarithmic functions instead. You can indirectly test your implementation with the `inspect` subcommand by sampling the integrand from within the unit hypercube, i.e. in `x_space` (below using an overall scale of 10 m_S for the conformal maps):

```
python3 triangler.py -param spherical inspect --x_space -p 0.1 0.2 0.3
{...} : +1.4783327522266039e+05 (excl. jacobian = +1.9153133080870564e-04)
```

with the input variables x_1 controling the radius, x_2 the cosine of the azimuthal angle and x_3 the polar angle. Cartesian parameterisation can be tested similarly with the `-param cartesian` option. Note that depending on the details of your implementation you may not reproduce exactly the result above. This is fine so long as the numerical integration still converges to the correct result in the later parts of this exercise.

2.6 Before actually integrating our LTD expression, it is interesting to plot the integrand. The `plot` function is already mostly implemented for you, but you must properly implement its call in the `main` function. Then find out the proper way of calling that subcommand to obtain a plot that will support your answer these two questions (and do not forget to specify the corresponding commands in the file `commands.makefile` of your submission):

- *2.6.a* Describe the region in momentum space seems to receive the largest contribution?
- *2.6.b* Which variable in x-space is the integrand most sensitive to? Around which value does it seem to be maximal?

2.7 [OPTIONAL] Implement support for 3D plots (e.g. by using the `mpl_toolkits.mplot3d` function of `matplotlib`). You can then use the `plot` subcommand with the option `-3D` to obtain a 3D plot of the integrand.

2.8 We are now ready to implement the integrator. Build a simple naive integrator (without any importance sampling) in the `naive_integrator` function. The simplest Monte-Carlo estimator of the central value of the integral is $< I >= \frac{1}{N}\sum_{i=1}^{N} w_i$ and of its error (directly deduced from the square root of its variance σ^2) is $< \Delta I >= \sqrt{\frac{<\sigma^2>}{N}} = \sqrt{\frac{\sum_{i=1}^{N} w_i^2/N - <I>^2}{N}}$. The integrand function to integrate is called `integrand_xspace`. Make sure to populate all fields of the object `IntegrationResult` returned by the function (read the functions defined in that class as they can prove to be useful for your implementation). You can test your implementation with the `integrate` subcommand:

```
./triangler.py -param spherical integrate -n 10 -ppi 10000 \
                        -it naive -nc 1 -s 1337
{...}
| > Integration result after 100000 evaluations in 0.89 CPU-s (8.9 µs / eval)
| > Max weight encountered = 4.70280e+01 at xs = [1.4346293742677796e-01 {...}]
| > Central value : +8.0764146899228422e+00   +/- 3.29e-02     (0.408%)
| >     vs target : +8.0863564465181099e+00   d = -9.94e-03    (-0.123%  = 0.30s)
```

Explain what are the quantities shown in the output and their meaning. Run the above again using the cartesian parameterisation and compare the variance. What do you observe? Why?

2.9 [OPTIONAL] Implement multicore parallelisation in your naive integrator (e.g. using `multiprocessing.Pool`) for the multiple calls to `integrand_xspace` and test it with the `--n_core 8` option. You will observe negligible improvement because the LTD integrand function is extremely simple in this case so that the rest of the non-parallelized code (e.g. parameterisation) is the bottleneck. You can however add an artificial `time.sleep(0.1)` statement in `integrand_xspace` in order to clearly see improvement from the parallelization (do not forget to remove it afterwards!).

2.10 Adaptive importance sampling is a common improvement to naive Monte-Carlo integration. The idea is to sample more densely the regions of the integrand where it is largest. This strategy is implemented in multiple public libraries such as `https://pypi.org/project/vegas`. The code provided already has a complete implementation using it in the function `vegas_integrator`. Find out the options for using this integrator using the help of the `integrate` subcommand and explain what is shown in the output. Report and compare the variance obtained in this case with the one obtained using your naive integrator.

2.11 [OPTIONAL] Multi-channeling is another common variance reduction technique. It consists in splitting the integrand into multiple channels, each of which is integrated separately with an appropriate optimised parameterisation (see additional information given during the exercise session). The total integral is then obtained by summing the results of each channel. Implement this technique for our LTD integrand in order to mitigate the impact on the variance introduced by the $\frac{1}{E_1 E_2 E_3}$ prefactor. You build the multi-channeled integrand by identically multiplying the integrand with $\frac{E_1^{-\alpha}+E_2^{-\alpha}+E_3^{-\alpha}}{E_1^{-\alpha}+E_2^{-\alpha}+E_3^{-\alpha}} = 1$, where α is a free hyper-parameter that can be optimized. You must implement this in the function `integrand_xspace` and then use the `integrate` subcommand with the option `-mc` to test it. For this simple integrand, you should find a marginal but measurable improvement in the variance. You should also ideally observe a reduction in the maximum weight encountered.

2.12 [OPTIONAL] The library `https://symbolica.io/docs/numerical_integration.html` offers powerful computer algebra functions, but also numerical integration routines including adaptive sampling. Besides a convenient interface and workflow offering maximum flexibility to the user regarding how they want to steer the numerical integration, it also offers the possibility of having a discrete important sampling over the integration channels. Implement the `Symbolica` integrator in the function `symbolica_integrator` and compare the resulting variance with that obtained with the other integrators, especially when using multi-channeling.

2.13 [Final showdown: OPTIONAL] The presence of the loop propagator mass m_ψ significantly smoothened the LTD integrand, especially the prefactor $\frac{1}{E_1 E_2 E_3}$. It would be interesting to also compute this triangle integral in a case where $m_\psi = 0$. However, in that situation, one faces the E-surfaces singularities discussed in Ex. 1.5. The solution to their numerical regularisation is very interesting, although beyond the scope of this exercise. However, we can still choose to set $m_\psi = 0$ and work within the unphysical Euclidean regime where $p^2 < 0, q^2 < 0, (p+q)^2 < 0$ so as to avoid these singularities. You can verify that the following integral indeed converges:

```
python3 triangler.py -param spherical --m_s 1. --m_psi 0. \
-p 0. 0. 0. 5. -q 1. 4. 3. 2. integrate -n 10 -ppi 10000 -it naive -nc 1 -s 1337
{...}
| > Integration result after 100000 evaluations in 0.84 CPU-s (8.4 µs / eval)
| > Max weight encountered = 1.10389e-02 at xs = [3.5194430845095082e-01 {...}]
| > Central value : +3.8647388792273368e-04   +/- 2.23e-06     (0.577%)
```

although our target analytical expression is not valid in this regime.

A useful figure of merit for the efficiency of an implementation of a numerical integration algorithm is the (square-root) of the variance estimator normalized to the central value of the integral, that is: $\bar{\sigma}(I) = \sqrt{N}\frac{<\Delta I>}{<I>}$. The goal of this final challenge is to optimize your implementation by adjusting hyperparameters, improving integrators, customising multi-channel definitions or even coming up with custom parameterisations so as to minimize $\bar{\sigma}(I)$. The subcommands `plot` and `inspect` (with additonal debug printouts and typically run on the maximum weight point reported after integration) can be useful when investigating such optimizations.

You will be ranked according to $\bar{\sigma}(I_1) + \bar{\sigma}(I_2) + \bar{\sigma}(I_3)$ where I_1 is the integral with the original physical parameters (i.e. that of Ex. 2.8), I_2 is the integral with the parameters above, and I_3 an integral with parameters kept secret to you so as to avoid your solution to be good solely thanks to overfitting (thus encouraging you to come up with generically applicable improvements).

Specify what command line you wish to be used when testing (include it in the `commands.makefile` file of your submission) and report what $\bar{\sigma}(I_1)$ and $\bar{\sigma}(I_2)$ you obtained when running this command yourself. The total number of sample points your command induces must lie between 10^6 and 10^7 sample points. The winner will be awarded an additional 10% bonus to their grade.

Q&A Think critically! In light of the above, include in your report any questions you may have regarding this methodology, its shortcomings and obstacles you may foresee when applying it to more complex integrals of actual interest in high-energy physics. These will be collected and reviewed together the week after its submission. (this part is not graded)