Eur. Phys. J. C (2018) 78:478
https://doi.org/10.1140/epjc/s10052-018-5948-0

THE EUROPEAN PHYSICAL JOURNAL C

CrossMark

Special Article – Tools for Experiment and Theory

PARTONS: PARtonic Tomography Of Nucleon Software

A computing framework for the phenomenology of Generalized Parton Distributions

B. Berthou[1], D. Binosi[2], N. Chouika[1], L. Colaneri[3,4], M. Guidal[3], C. Mezrag[5], H. Moutarde[1,a], J. Rodríguez-Quintero[7,8], F. Sabatié[1], P. Sznajder[3,6], J. Wagner[6]

[1] IRFU, CEA, Université Paris-Saclay, 91191 Gif-sur-Yvette, France
[2] ECT*/Fondazione Bruno Kessler, Villa Tambosi, Strada delle Tabarelle 286, 38123 Villazzano, TN, Italy
[3] Institut de Physique Nucléaire d'Orsay, CNRS-IN2P3, Université Paris-Sud, Université Paris-Saclay, 91406 Orsay, France
[4] University of Connecticut, Storrs, CT 06269, USA
[5] Istituto Nazionale di Fisica Nucleare, Sezione di Roma, P. le A. Moro 2, 00185 Roma, Italy
[6] National Centre for Nuclear Research (NCBJ), 00-681 Warsaw, Poland
[7] Dpto. Ciencias Integradas, Centro de Estudios Avanzados en Fis., Mat. y Comp., Fac. Ciencias Experimentales, Universidad de Huelva, 21071 Huelva, Spain
[8] CAFPE, Universidad de Granada, 18071 Granada, Spain

Received: 3 April 2018 / Accepted: 29 May 2018 / Published online: 11 June 2018

Abstract We describe the architecture and functionalities of a C++ software framework, coined PARTONS, dedicated to the phenomenology of Generalized Parton Distributions. These distributions describe the three-dimensional structure of hadrons in terms of quarks and gluons, and can be accessed in deeply exclusive lepto- or photo-production of mesons or photons. PARTONS provides a necessary bridge between models of Generalized Parton Distributions and experimental data collected in various exclusive production channels. We outline the specification of the PARTONS framework in terms of practical needs, physical content and numerical capacity. This framework will be useful for physicists – theorists or experimentalists – not only to develop new models, but also to interpret existing measurements and even design new experiments.

1 Introduction

Generalized Parton Distributions (GPDs) were independently discovered in 1994 by Müller et al. [1] and in 1997 by Radyushkin [2] and Ji [3]. This subfield of Quantum Chromodynamics (QCD) grew rapidly because of the unique theoretical, phenomenological and experimental properties of these objects. GPDs are related to other non-perturbative QCD quantities that were studied previously without any connection: Parton Distribution Functions (PDFs) and Form Factors (FFs). In an infinite-momentum frame, where a hadron is flying at near the speed of light, PDFs describe the longitudinal momentum distributions of partons inside the hadron and FFs are the Fourier transforms of the hadron charge distributions in the transverse plane. PDFs and FFs appear as limiting cases of GPDs, which, among many other important properties on the hadron structure, encode the correlation between longitudinal momentum of partons and their transverse plane position. In the pion case GPDs also extend the notion of Distribution Amplitudes (DA), which probe the two-quark component of the light cone wave function. This generality is complemented by one remarkable feature: the GPDs of a given hadron are directly connected to the matrix elements of the QCD energy-momentum tensor evaluated between adequate momentum states of the corresponding hadron. More precisely, those matrix elements can be paramaterized in terms of Mellin moments of GPDs. This is both welcome and unexpected because the energy-momentum tensor is canonically probed through gravity. GPDs bring the energy-momentum matrix elements within the experimental reach through electromagnetic scattering. Indeed GPDs themselves – hence their Mellin moments – are accessible in facilities running experiments with lepton beams.

It was realized from the early days that the leptoproduction of a real photon off a nucleon target, referred to as Deeply Virtual Compton Scattering (DVCS), is the theoretically cleanest way to access GPDs. At the beginning of the twenty first century, first measurements of DVCS were reported by the HERMES [4] and CLAS [5] collaborations, establishing the immediate experimental relevance of the concept and marking the beginning of the experimental era of this field. Several

[a] e-mail: herve.moutarde@cea.fr

dedicated experiments and sophisticated theoretical developments followed, putting the field in a good shape as many reviews testify [6–14].

GPDs are natural extensions of PDFs and yet their phenomenology is much harder. The lack of a general first principles parameterization justifies the need for several models, while a large number of possibly involved GPDs requires a multichannel analysis to constrain them from various experimental filters. GPDs belong to an active research field where deep theoretical questions are to be solved, in conjunction with existing experimental programmes, technological challenges, computational issues, as well as well-defined entities and measurements. The foreseen accuracy of experimental data to be measured at Jefferson Lab [15] and at COMPASS [16] requires the careful design of tools to meet the challenge of the high-precision era, and to be able to make the best from experimental data. The same tools should also be used to design future experiments or to contribute to the physics case of the foreseen Electron Ion Collider (EIC) [17] and Large Hadron Electron Collider (LHeC) [18]. Integrating those tools in one single framework is the aim of the PARTONS project.

The paper is organized as follows. The second section is a reminder of the phenomenological framework: how GPDs are defined, and how they can be accessed experimentally. We will illustrate the discussion with the example of DVCS. Then, we discuss the need assessments for high precision GPD phenomenology in the third section. The fourth section describes the code architecture, while the fifth one lists existing modules. The sixth section provides several examples.

2 Phenomenological framework

We will now shortly review the main building blocks of the description of exclusive processes, starting from the definition of GPDs, through the cross section calculations with the use of coefficient functions and Compton Form Factors (CFFs), up to the definition of various observables. The structure of such a calculation, described on the example of DVCS, determines the structure of the PARTONS framework.

2.1 Definition of Generalized Parton Distributions

Unpolarized quark (superscript q) or gluon (superscript g) GPDs of a spin-1/2 massive hadron (of mass M) are defined in the light cone gauge by the following matrix elements:

$$F^q(x,\xi,t) = \frac{1}{2}\int \frac{dz^-}{2\pi} e^{ixP^+z^-} \times \left\langle P+\frac{\Delta}{2}\middle| \bar{q}\left(-\frac{z}{2}\right)\gamma^+ q\left(\frac{z}{2}\right)\middle| P-\frac{\Delta}{2}\right\rangle\Big|_{\substack{z^+=0\\ z_\perp=0}}, \quad (1)$$

$$F^g(x,\xi,t) = \frac{1}{P^+}\int \frac{dz^-}{2\pi} e^{ixP^+z^-} \times \left\langle P+\frac{\Delta}{2}\middle| G_a^{+\mu}\left(-\frac{z}{2}\right) G_{a\mu}^{\ +}\left(\frac{z}{2}\right)\middle| P-\frac{\Delta}{2}\right\rangle\Big|_{\substack{z^+=0\\ z_\perp=0}}. \quad (2)$$

We note $\xi = -\Delta^+/(2P^+)$ the skewness variable and $t = \Delta^2$ the square of the four-momentum transfer on the hadron target. We adopt conventions of Ref. [8], and the superscript "+" refers to the projection of a four-vector on a light-like vector n_+. The average momentum P obeys $P^2 = M^2 - t/4$. Analogous definitions for the polarized quark and gluon GPDs $\widetilde{F}^{q,g}$ can be found in Ref. [8].

Both F^a and $\widetilde{F}^a$ ($a = q, g$) can be decomposed as:

$$F^a(x,\xi,t) = \frac{1}{2P^+}\left(h^+ H^a(x,\xi,t) + e^+ E^a(x,\xi,t)\right), \quad (3)$$

$$\widetilde{F}^a(x,\xi,t) = \frac{1}{2P^+}\left(\tilde{h}^+ \widetilde{H}^a(x,\xi,t) + \tilde{e}^+ \widetilde{E}^a(x,\xi,t)\right), \quad (4)$$

where the Dirac spinor bilinears are:

$$h^\mu = \bar{u}\left(P+\frac{\Delta}{2}\right)\gamma^\mu u\left(P-\frac{\Delta}{2}\right), \quad (5)$$

$$e^\mu = \frac{i\Delta_\nu}{2M}\bar{u}\left(P+\frac{\Delta}{2}\right)\sigma^{\mu\nu} u\left(P-\frac{\Delta}{2}\right), \quad (6)$$

$$\tilde{h}^\mu = \bar{u}\left(P+\frac{\Delta}{2}\right)\gamma^\mu\gamma_5 u\left(P-\frac{\Delta}{2}\right), \quad (7)$$

$$\tilde{e}^\mu = \frac{\Delta^\mu}{2M}\bar{u}\left(P+\frac{\Delta}{2}\right)\gamma_5 u\left(P-\frac{\Delta}{2}\right), \quad (8)$$

allowing for the identification of four GPDs: H, E, $\widetilde{H}$ and $\widetilde{E}$. The spinors are normalized so that $\bar{u}(p)\gamma^\mu u(p) = 2p^\mu$.

In principle, GPDs depend on a renormalization scale μ_R and a factorization scale μ_F, which are usually set equal to each other. From the point of view of code writing, we however keep two different variables representing the scales, even though we have taken them equal in all applications so far.

2.2 Experimental access to Generalized Parton Distributions

GPDs are accessible in hard exclusive processes, where properties of all final state particles are reconstructed, and existence of hard scale allows for the factorization of amplitudes into GPDs and perturbatively calculable coefficient functions. Three exclusive channels attract most of the current experimental interest: Deeply Virtual Compton Scattering (DVCS), Timelike Compton Scattering (TCS) [19] and Deeply Virtual Meson Production (DVMP) [20]. However, also other ones, like Double Deeply Virtual Compton Scattering (DDVCS) [21,22], Heavy Vector Meson Production (HVMP) [23], two particle production [24,25] and neutrino-induced exclusive reactions [26–28], may be necessary to provide the full picture of hadron structure.

The pioneering DVCS measurements at the beginning of the twenty first century had been followed by numerous dedicated experimental campaigns [29–49]. During the same period, an intense theoretical activity put DVCS under solid control. In particular we mention the full description of DVCS up to twist-3 [50–53], the computation of higher orders in the perturbative QCD expansion [54–63], the soft-collinear resummation of DVCS [64,65], the discussion of QED gauge invariance [66–70] and the elucidation of finite-t and target mass corrections [71,72]. Variety of those existing theoretical improvements, usually developed within the model/framework preferred by the corresponding authors, also justify the need for a common framework enabling systematic comparisons.

2.2.1 *Theory of Deeply Virtual Compton Scattering*

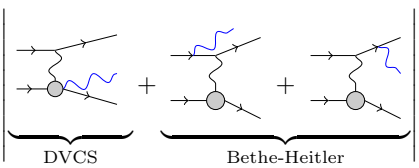
A typical evaluation of cross sections involving GPDs is illustrated here on the most prominent example of exclusive process, *i.e.* the lepto-production of a real photon on a nucleon target N:

$$l(k,h_l)+N(p,h)\to l(k',h_l')+N(p',h')+\gamma(q',\lambda')\ , \quad (9)$$

where the first letters in parentheses are the four-momenta, while the second ones are the helicities of the particles. The amplitude $\mathcal{T}$ for this process is the coherent superposition of the DVCS and Bethe-Heitler (BH) amplitudes:

$$|\mathcal{T}|^2=|\mathcal{T}_{\rm BH}+\mathcal{T}_{\rm DVCS}|^2=|\mathcal{T}_{\rm BH}|^2+|\mathcal{T}_{\rm DVCS}|^2+\mathcal{I}, \quad (10)$$

with $\mathcal{I}$ standing for the interference between BH and DVCS processes. In terms of Feynman diagrams one has:

$\sigma(ep\to ep\gamma)\sim$ 

The BH amplitude is under very good control since it can be computed in perturbative Quantum Electrodynamics, and because it depends on the experimentally well-known nucleon FFs. We note $q=k-k'$ the four-momentum of the virtual photon in DVCS, and:

$$Q^2=-q^2\ , \quad (11)$$

$$x_B=\frac{Q^2}{2\ p\cdot q}\ , \quad (12)$$

$$t=(p-p')^2\ . \quad (13)$$

The corresponding cross-section is five-fold differential in x_B, Q^2, t and two azimuthal angles. These are the angle ϕ between the lepton scattering plane and the production plane (spanned by the produced photon and nucleon momenta), and the angle ϕ_S between the lepton scattering plane and the target spin component perpendicular to the direction of the virtual photon, see Fig. 1.

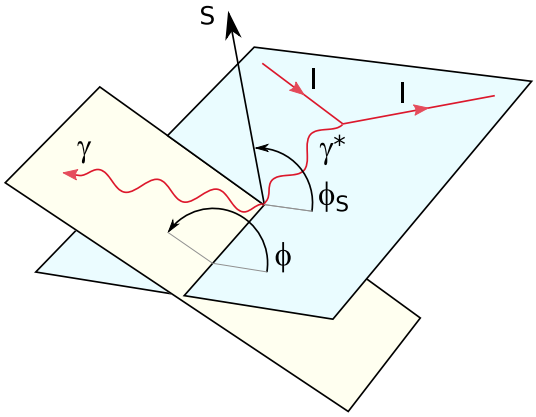


Fig. 1 Kinematics of DVCS in the target rest frame. ϕ is the angle between the leptonic plane (spanned by the incoming and outgoing lepton momenta), and the production plane (spanned by the outgoing photon and nucleon momenta). ϕ_S denotes the angle between the nucleon polarization vector and the leptonic plane

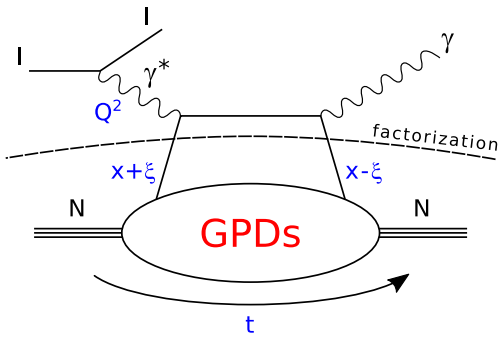


Fig. 2 Partonic interpretation of the DVCS process

2.2.2 *Factorization of DVCS and coefficient functions*

The Bjorken limit, defined by:

$$Q^2\to\infty \text{ at fixed } x_B \text{ and } t\ , \quad (14)$$

ensures the factorization for the DVCS amplitude [2,57,73,74], which provides a *partonic* interpretation of the *hadronic* process: it is possible to reduce the reaction mechanism to the scattering of a virtual photon on one *active* parton. Such an interpretation at Leading Order (LO) is presented in Fig. 2.

The DVCS amplitude $\mathcal{T}_{\rm DVCS}$ can be decomposed either in twelve helicity amplitudes or, equivalently, in twelve Compton Form Factors (CFFs), which are usually denoted as $\mathcal{H}$,

$\mathcal{E}$, $\widetilde{\mathcal{H}}$, $\widetilde{\mathcal{E}}$, $\mathcal{H}_3$, $\mathcal{E}_3$, $\widetilde{\mathcal{H}}_3$, $\widetilde{\mathcal{H}}_3$, $\mathcal{H}_\mathrm{T}$, $\mathcal{E}_\mathrm{T}$, $\widetilde{\mathcal{H}}_\mathrm{T}$, $\widetilde{\mathcal{E}}_\mathrm{T}$, with symbols reflecting their relation to GPDs. The last eight CFFs are related to the twist-three ($\mathcal{F}_3$) and transversity ($\mathcal{F}_\mathrm{T}$) GPDs, and usually disregarded in present analyses of DVCS data as subdominant contributions.

To keep the discussion simple, we will now focus on the GPD H and the associated CFF $\mathcal{H}$. After a proper renormalization, the CFF $\mathcal{H}$ reads in its factorized form (at factorization scale μ_F):

$$\mathcal{H} = \int_{-1}^{1} \mathrm{d}x \left[\sum_{q}^{N_f} T^q(x) H^q(x) + T^g(x) H^g(x) \right] , \quad (15)$$

where the explicit ξ and t dependencies are omitted, and N_f is the number of active quark flavors. The renormalized coefficient functions are given by:

$$T^q(x) = \left[C_0^q(x) + C_1^q(x) + \ln\left(\frac{Q^2}{\mu_F^2}\right) C_{\mathrm{coll}}^q(x) \right] - (x \to -x) , \quad (16)$$

$$T^g(x) = \left[C_1^g(x) + \ln\left(\frac{Q^2}{\mu_F^2}\right) C_{\mathrm{coll}}^g(x) \right] + (x \to -x) . \quad (17)$$

We only show the coefficient function being the result of LO calculation:

$$C_0^q(x,\xi) = -e_q^2 \frac{1}{x + \xi - i\epsilon}, \quad (18)$$

where e_q is the quark electric charge in units of the positron charge. We refer to the literature for the Next-to-Leading Order (NLO) coefficient functions $C_1^{q,g}$ and $C_{\mathrm{coll}}^{q,g}$ [54,56–62].

2.2.3 Observables of the DVCS channel

The cross section of electroproduction of a real photon off an unpolarized target can be written as:

$$\mathrm{d}\sigma^{h_l,e_l}(\phi) = \mathrm{d}\sigma_{\mathrm{UU}}(\phi) \left[1 + h_l A_{\mathrm{LU,\,DVCS}}(\phi) + e_l h_l A_{\mathrm{LU,\,I}}(\phi) + e_l A_{\mathrm{C}}(\phi) \right] , \quad (19)$$

where e_l is the beam charge (in units of the positron charge) and $h_l/2$ the beam helicity. If longitudinally polarized, positively and negatively charged beams are available, the asymmetries in Eq. (19) can be isolated. This is the case for a large part of the data collected by HERMES. For example, the beam charge asymmetry is obtained from the combination:

$$A_{\mathrm{C}}(\phi) = \frac{1}{4\mathrm{d}\sigma_{\mathrm{UU}}(\phi)} \left[(\mathrm{d}\sigma^{\overset{+}{\rightarrow}}(\phi) + \mathrm{d}\sigma^{\overset{+}{\leftarrow}}(\phi)) - (\mathrm{d}\sigma^{\overset{-}{\rightarrow}}(\phi) + \mathrm{d}\sigma^{\overset{-}{\leftarrow}}(\phi)) \right] , \quad (20)$$

where we denote by "±" the sign of the beam charge e_l, and by the arrow $\rightarrow$ ($\leftarrow$) the helicity plus (minus). From similar combinations, we obtain the two beam spin asymmetries $A_{\mathrm{LU,\,I}}$ and $A_{\mathrm{LU,\,DVCS}}$:

$$A_{\mathrm{LU,\,I}}(\phi) = \frac{1}{4\mathrm{d}\sigma_{\mathrm{UU}}(\phi)} \left[(\mathrm{d}\sigma^{\overset{+}{\rightarrow}}(\phi) - \mathrm{d}\sigma^{\overset{+}{\leftarrow}}(\phi)) - (\mathrm{d}\sigma^{\overset{-}{\rightarrow}}(\phi) - \mathrm{d}\sigma^{\overset{-}{\leftarrow}}(\phi)) \right] , \quad (21)$$

$$A_{\mathrm{LU,\,DVCS}}(\phi) = \frac{1}{4\mathrm{d}\sigma_{\mathrm{UU}}(\phi)} \left[(\mathrm{d}\sigma^{\overset{+}{\rightarrow}}(\phi) - \mathrm{d}\sigma^{\overset{+}{\leftarrow}}(\phi)) + (\mathrm{d}\sigma^{\overset{-}{\rightarrow}}(\phi) - \mathrm{d}\sigma^{\overset{-}{\leftarrow}}(\phi)) \right] . \quad (22)$$

If an experiment cannot change the value of the electric charge of the beam (such as in Jefferson Lab), the asymmetries defined in Eq. (19) cannot be isolated anymore, and one can only measure the following (total) beam spin asymmetry $A_{\mathrm{LU}}^{e_l}$:

$$A_{\mathrm{LU}}^{e_l}(\phi) = \frac{\mathrm{d}\sigma^{\overset{e_l}{\rightarrow}}(\phi) - \mathrm{d}\sigma^{\overset{e_l}{\leftarrow}}(\phi)}{\mathrm{d}\sigma^{\overset{e_l}{\rightarrow}}(\phi) + \mathrm{d}\sigma^{\overset{e_l}{\leftarrow}}(\phi)}. \quad (23)$$

This definition of $A_{\mathrm{LU}}^{e_l}$ can be expressed as a function of the spin and charge asymmetries defined in Eq. (19):

$$A_{\mathrm{LU}}^{e_l}(\phi) = \frac{e_l A_{\mathrm{LU,\,I}}(\phi) + A_{\mathrm{LU,\,DVCS}}(\phi)}{1 + e_l A_{\mathrm{C}}(\phi)} . \quad (24)$$

We refer to Ref. [75] for a systematic nomenclature of DVCS observables and their relations to CFFs. Because different observables are related to different combinations of CFFs with different weighting factors, a flexible code for the phenomenology of GPDs should not only be able to deal with different exclusive channels, but also with cross sections and various asymmetries. This is one of the main constraints on the design of the PARTONS framework.

3 Needs assessment

3.1 From GPDs to observables: basic structure

The basic structure of the computation of an observable of one channel related to GPDs is outlined in Fig. 3. We illustrate the situation in the DVCS case, but the following considerations should apply to any channel. The *large distance* level contains GPDs as functions of x, ξ, t, μ_F and μ_R, which in addition are dependent on unspecified (model-dependent) parameters. The dependence on the factorization scale μ_F

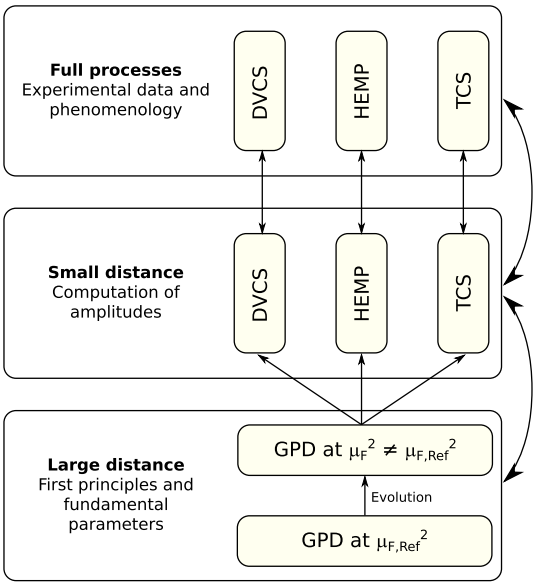


Fig. 3 The computation of an observable in terms of GPDs is generically layered in three basic steps: description of the hadron structure with nonperturbative quantities, computation of coefficient functions, and evaluation of cross sections

is described by evolution equations. The kernels of the GPD evolution equations at LO were derived in the seminal papers introducing GPDs or soon after [1,3,73,76,77]. The kernels at NLO were obtained in Refs. [78–82]. The corresponding work for transversity GPDs was published in Refs. [83–85]. To stay as generic as possible, evolution equations should be solved mostly in x-space, but with different numerical integration routines, if we require either speed and/or accuracy. The *small distance* level convolutes GPDs with various coefficient functions depending on the considered channel (see Sect. 2.2.2). Again, at this point we should be free to select the integration routine fulfilling our needs. Various theoretical frameworks exist that take into account *e.g.* the target mass and finite-t corrections [71,72], the soft-collinear resummation of DVCS [64,65], higher order effects either in the coefficient function [54–58] or in the evolution kernel [78–82]. All theoretical frameworks should work all the same with a given GPD model. The *full process* level produces cross sections or asymmetries (see Sect. 2.2.3) for various kinematics. For fitting purposes, all observables (whatever the channel is) should be treated in the same manner in order to simplify handling of experimental data. We may want to check *e.g.* the impact of one specific data set on the general knowledge of GPDs, or to apply some kinematic cuts in order to guarantee that the analysis takes place in a range where factorization theorems apply. Note, that if we want to fit data (say, if we want to minimize a χ^2 value), then we will have to loop over such GPD-to-observables structure at each step of the minimization.

3.2 Needs and constraints

The basic structure of the computations, the type of studies to be done, or simply the profile of the users, already put strong constraints on the software architecture design.

First of all, maintaining the software framework, or adding new theoretical developments (*e.g.* the aforementioned recent computation of target mass and finite-t corrections) should be as easy as possible. The structure of the framework should be flexible enough to allow the manipulation of an important number of physical concepts of a different nature. For instance, we may want to use the same tools to test new theoretical ideas and to design new experiments. Implicitly, the user of the code will probably know only remotely the detailed description of the physical module he is using – we cannot expect any user to be an expert in any physical model involved in the framework. However, a careful user should always get a correct result, even without knowing all details of the implementation. This means that all that *can be* automated *has to be* automated, and that physical modules should be designed in such a way that an inadequate use is forbidden, or indicated to the user with an explicit warning.

Second, with respect to maintenance, we want to be sure that adding new functionalities or new modules will not do any harm to the existing pieces of code. This requires some non-regression tools to guarantee that the version $n+1$ has (at least) all the functionalities of the version n. To trace back the results of the code (*e.g.* to be able to reproduce the results of a fit), it should be possible to save some *computing scenarios* for a later reference. The maintenance of the PARTONS code on a long-term perspective is one of the key element of its design, aimed at both the robustness and the flexibility. It was developed following *agile development procedures* structured in cycles, with intermediate deliverables and a functioning architecture all way long.

Third, the code should ideally be used by a heterogeneous population of users, ranging from theoretical physicists used to symbolic computation softwares, to experimentalists using the CERN library ROOT [86,87] and Monte Carlo techniques to design new experiments.

Fourth, the code should produce outputs of various kinds. As mentioned above, it should be able to deal with any kind of conceivable observables related to exclusive processes. From the software design point of view, all types of observables should be described in a generic way to simplify the selection and manipulation of data. This in particular would greatly simplify future global fits of experimental data. However, cross sections and asymmetries are very complicated outputs, which integrate a lot of physical hypothesis and mathematical techniques. To properly estimate the importance of a given physical assumption or a numerical routine accuracy, it is necessary to handle intermediate outputs, like GPDs them-

selves and CFFs. The modular structure of the PARTONS framework makes it possible. The output of each module is a well-defined object that can be stored in a database, if requested by the user running the code. Requests to the database allow post-processing of the data, either through plots or data tables.

Ideally, it should have been possible to run the code through a web interface, in the spirit of `Durham service` for PDFs [88,89]. However, such a solution requires dedicated work to synchronize a database to the web page, to prevent it of any attack, to create a queue system if several users want to perform their computations at the same time and to handle large volumes of data, which can always happen with functions depending on (at least) three `double` variables x, ξ and t. In particular, this means that a dedicated engineer has to take part of his time to tackle these problems and maintain all basic features. For now we let users download a client application to run the code on their own machines. We offer two possibilities – one can either download the source code of PARTONS and compile it by oneself, or download a preconfigured appliance of a virtual machine. The first way requires the availability of additional libraries, see Sect. 4, but in particular it allows to have PARTONS at computing farms. The second way only requires `VirtualBox` [90], which is one of the most popular virtualization suites. Our provided virtual machine allows to run PARTONS as it was out-of-the-box, independently of the user's operating system.

Finally, let us mention that the field of 3D hadron structure has been witnessing in parallel a similar collaborative effort for the phenomenology of Tranverse Momentum Dependent parton distribution functions (TMDs): the `tmdlib` [91,92] library offers an interface to various TMD models. In our view, the complexity of each of these fields, and their respective needs and timescales, has fully justified the development of two independent GPD and TMD projects. However, since both projects have become mature enough, the natural discussions between the two communities will provide a very valuable feedback.

4 Code architecture

The PARTONS framework is written in C++. This choice has been made for performances and to have a homogeneous product in terms of coding and programming languages. In particular, there is no wrapping of other third-party softwares written in a different programming language. The project considers two different communities: the developers, who have to understand the software architecture to use low-level functions, and the users, who can just use high-level functions ignoring the details of implementations. With the progress of automation, the users may run the code without writing a line of C++ code. In the community of developers, a crucial role is played by the software architect, who is responsible for the integration of new modules in the framework. He guarantees the robustness and homogeneity of the code being developed. We have decided to depend as little as possible on third-party libraries to help its dissemination. Presently, the PARTONS code contains only one residual dependency on the `CLN` library [93], that is needed for one particular GPD module and can be easily suppressed later. Only the dependence on the cross-platform application framework `Qt` [94] is essential, because it manages the connections to different types of databases in a generic way (see Sect. 4.5). The `SFML` library [95] is also needed to handle threads (see Sect. 4.6). Information about the licenses is given in Sect. 7.

From the software engineering point of view, the PARTONS project benefits from a layered and service-oriented architecture, which provides both the flexibility and the standardization. To the best of our knowledge, this architecture is original in the world of scientific computing, at least in nuclear and particle physics. It is derived from web-oriented technologies, such as the `Java EE` specification [96]. We describe below the whys and hows of these choices.

4.1 Layers

Ideally, the code should not have to go through a major rewriting during the years dedicated to the analysis of Jefferson Lab and COMPASS exclusive data. One way to ensure this is to isolate potential modifications as well as possible. This is the reason for the layered architecture: every part of the architecture belongs to a *layer*, and a modification in one layer does not hinder other layers.

The layered structure of the PARTONS software is shown in Fig. 4 and is made of seven parts. The Module layer is a collection of single encapsulated developments of various types. This layer contains the physics engine, like GPD models, but also computations of coefficient functions and cross sections with various physics assumptions. A module is fed by data and it produces results, which corresponds to the Data and Result layers, respectively. In these two layers no treatment is made on data. These are just collections of containers (high-level objects) that make sure, for example, that each module receives an object that has been well-formed thanks to its constructor. For instance, instead of feeding a GPD module with 5 `double` variables (x, ξ, t, μ_F^2 and μ_R^2), which can be sent in an incorrect order after some minor editing of the code, we isolate all places where such kind of errors may happen. We trust the fact that the high-level object (here `GPDKinematic`) has been correctly constructed from those five `double` variables. The risk of an accidental manipulation (*e.g.* an exchange of μ_F^2 and t) becomes much more limited. To illustrate this, we provide the code defining the `GPDKinematic` class:

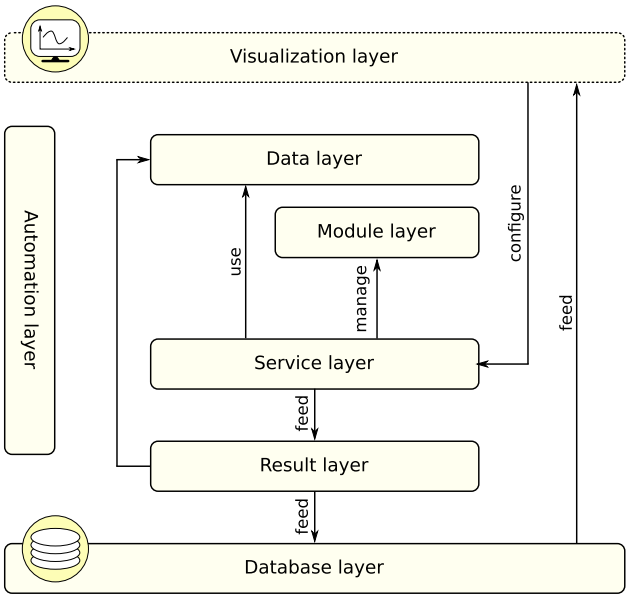


Fig. 4 The layered structure of the PARTONS framework. The Visualization layer, allowing users to launch computations from a visualizing interface, is not available in the first release of PARTONS

```
class GPDKinematic: public Kinematic {

public:

    // Default constructor
    GPDKinematic();

    // Assignment constructor
    GPDKinematic(double x, double xi, double t, double MuF2, double MuR2);

    // Constructor for automation
    GPDKinematic(ParameterList &parameterList);

    // Return pre−formatted characters string containing private
    // member values
    virtual std::string toString() const;

    // Getters and setters of private members values
    ...

private:

    // Longitudinal momentum fraction of the active parton
    double m_x;

    // Skewness
    double m_xi;

    // Squared four−momentum transfer between initial and final
    // hadron (in GeV^2)
    double m_t;

    // Squared factorization scale (in GeV^2)
    double m_MuF2;

    // Squared renormalization scale (in GeV^2)
    double m_MuR2;
};
```

The class contains `double` variables used to store the value of x, ξ, t, μ_F^2 and μ_R^2, and three methods to:

- create a new `GPDKinematic` object from a set of five `double` variables,
- create it from a generic list of parameters encoded in the `ParameterList` container (used by the automation),
- return a `std::string` variable containing an alphanumeric representation of the object to be used *e.g.* to print its content on a screen or in a file.

We emphasize that this structure is not specific to GPD modules. Every family of modules has its own input and output types, which are generically referred to as the *beans*. Storing all input variables in simple high-level objects also makes sure that, for example, a GPD model will not accidentally be evaluated at something completely different, such as an angular variable (still a `double` variable), which also appears in a DVCS kinematic configuration.

Another critical element of the architecture is the Service layer being a collection of services. The services link related modules to offer high-level functions to the users, and help hide the complexity of low-level functions. The whole code can be used without the services, however it is less convenient.

At last, three extra layers provide useful functionalities to the users. The Database layer contains tools to store results in local or remote databases. It is designed to optimize later requests and post-processing treatments, and to limit data redundancies in the used databases. The Automation layer is a collection of tools designed for the purpose of automation. A scenario, *i.e.* XML file containing all physical and mathematical assumptions on the computation to be performed, is parsed. With the XML file parsed, all relevant objects are created and evaluations are processed. The results are shown at the standard output and/or they can be stored in a database, including the associated scenario, either to trace back all hypothesis underlying the results, or to be able to evaluate them again later, *e.g.* for non-regression purposes. Finally, the Visualization layer, which is not available in the first release of PARTONS, integrates all visualizing tools. With this layer, users will be able to make requests to the used databases containing the output data through an interface, and draw curves on a screen and/or produce grids of points in files.

4.2 Modules

The flexibility of the architecture is achieved through the class inheritance. The logical sequence of the code as a whole is centralized in classes that receive *standardized* inputs and return *standardized* outputs. All details of model descriptions, numerical precisions, *etc.*, are exclusively left to the child classes.

An example is provided in Fig. 5, which describes the actual implementation of GPD modules. The input is a `GPDKinematic` object described above. The output is an object called `GPDResult`, which contains GPDs provided

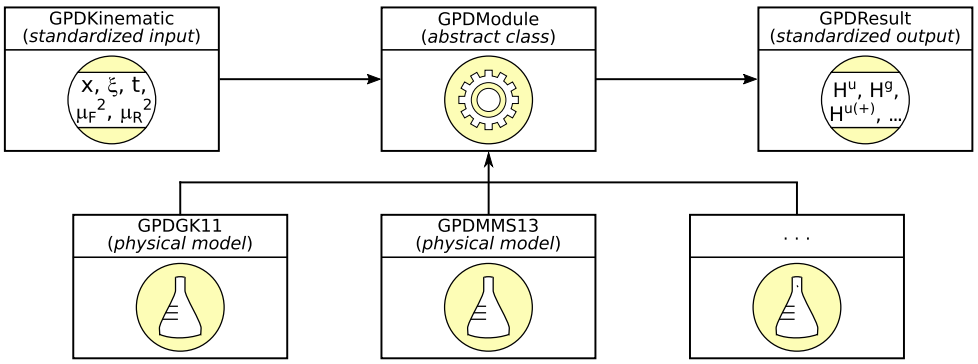


Fig. 5 Modularity through class inheritance and standardized inputs and outputs

by the considered model, with separate values for gluons and all available quark flavors, including singlet and non-singlet combinations. It also contains GPD kinematics and an identifier of the used GPD model, to trace back all conditions of the evaluation. Finally, `GPDResult` also contains functions to filter the data (*e.g.* depending on the parton type) or to print the results.

`GPDModule` is a collection of methods to compute various GPDs (*e.g.* H or E). There is no upper or lower limits on the number of GPD types that `GPDModule` should contain. A crucial part of the implementation of `GPDModule` class is shown here:

```
// Computes GPDs with input parameters
virtual PartonDistribution compute(double x, double xi, double t,
        double MuF2, double MuR2, GPDType::Type gpdType,
        bool evolution = true);

// This method can be implemented in the child class to make the GPD H available for computations
virtual PartonDistribution computeH();
```

Here, the variable `gpdType` selects the type of GPD to be computed, *i.e.* H, E, ..., or all GPDs available in the considered model. The addition of a new GPD is fairly simple. It suffices to inherit from a class (`GPDModule` or even its children), and to implement only the appropriate "compute" functions, *e.g.* `computeHt` for the GPD $\widetilde{H}$. Such a new child class contains all specific implementation corresponding to *e.g.* the GK [97–99] or VGG [7,66,100,101] models. Any model obeying this general structure can enter the PARTONS framework and benefit from all the other features.

The example discussed above for GPD models can be extended to any other types of modules, such as QCD evolution modules, DVCS observable modules, *etc.*

4.2.1 Registry

Adding a new child class does not require any modification of the existing code as long as this class inherits from an existing module. In particular, we can freely add as many GPD models as we want. On the contrary, if we wish to extend the functionalities of the PARTONS framework to the computation of *e.g.* TMDs, similarly to the `tmdlib` project [91], we will have to create all parent classes to define what TMDs are. Adding a new module simply consists in adding a new file to the whole project. The interoperability of the PARTONS structure is thus maintained all way long. This essential feature is provided by the *Registry*.

The Registry is the analog of a phone book, which lists all available modules. From the software engineering point of view, the Registry corresponds to the *singleton* design pattern, which ensures that it may exist in the memory only as a unique object. The modules are created and registered in the Registry at the beginning of the code execution, when `const static` variables are initialized in all classes, prior to the execution of the `main` code. Here is an example of such initialization:

```
#include "../../../../include/partons/BaseObjectRegistry.h"

// Initialize static const class_id member with a number
// returned by BaseObjectRegistry after a successful
// registration with a unique name
const unsigned int GPDGK11::classId = BaseObjectRegistry::getInstance()->
registerBaseObject(new GPDGK11("GPDGK11"));
```

During the execution of this code, the first thing to do is to call the unique instance of the Registry and to register the new module with the class name provided by the developer of the module. The underlying mechanism in illustrated by Fig. 6. If the new module is successfully registered, the Registry returns a unique identifier encoded in an `int` variable for performance purposes. This identifier is the same throughout the whole platform and for all instances of the module. The identifier being unique and registered prevents from an undesirable code operation. For example, if a user accidentally asks for a non-existent GK12 model (`GPDGK12::classId`) instead of GK11 (`GPDGK11::classId`), the code will simply not compile. This would not have been achievable, if modules were identified by a simple type such as `string`.

At this stage, it is important to mention that the Registry stores pointers to all modules in a generic way, *i.e.* whatever their nature is: pointers to `GPDModule`, to `RunningAlphaStrongModule`, *etc.* This is achieved by requiring all modules to derive from a single parent class named `BaseObject`. `BaseObject` is the "zeroth-level-object" of the architecture. Any C++ object in PARTONS

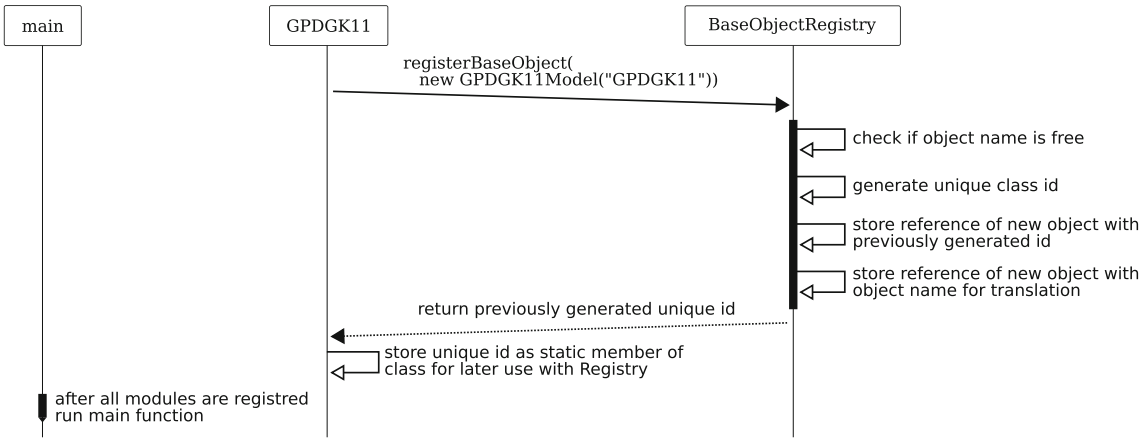


Fig. 6 Sequence diagram presenting the different steps, arranged in time, allowing the self-registration of all modules at the start of the execution of the PARTONS code. Parallel vertical lines (lifelines) represent the different processes or objects simultaneously living. Vertical black boxes indicate the duration of the process between the function call and return. Solid lines with open arrows indicate operations during the process, while those with filled arrows indicate function calls. Dotted lines with arrows indicate function returns

can and should inherit from it. It also carries information on the identity of a specific object, which can be transmitted as an explicit message to the Logger (see Sect. 4.6.2). This information is understandable to a human being, in contrary to an address in memory.

4.2.2 *Factory*

The Registry lists everything that is available in the platform, but only one species of each. If one wants to use a module, one cannot take it from the Registry, otherwise it would not be available anymore. The solution consists in using the *Factory* design pattern, which gives to the user a pre-configured copy of an object stored in the Registry. The user can then manage the configuration of the module and its life cycle.

The principle of the Factory is the following. We consider once again the example of `GPDGK11`. By construction, `GPDGK11` is derived from `GPDModule`, which itself is derived from `BaseObject` to be stored generically in the Registry. As shown in Fig. 7, when a user wants to use the GPD model identified by `GPDGK11::classId`, he asks `ModuleObjectFactory` to return him a pointer of `GPDModule` type. `ModuleObjectFactory` asks `BaseObjectFactory` to provide a new instance of `GPDModule` identified by `GPDGK11::classId`. To this aim, `BaseObjectFactory` requests that the Registry gives back the reference to `GPDGK11` already stored in the memory. The Registry goes through its internal list to find `BaseObject` with identifier `GPDGK11::classId`. Using the found reference, `BaseObjectFactory` clones[1] the `GPDGK11` object and provides `Module Object Factory` with a reference to the duplicated object. Finally, `ModuleObjectFactory` casts the pointer to this new object to the appropriate type `GPDModule`. What is needed to fit to the structure of the code is `GPDModule` (GPD models are all objects of the same type when seen from the exterior of a black box). The specific implementation *i.e.* what defines a single model from the physics point of view (and what is in the black box from the software point of view) is in a child class, like `GPDGK11`. Through pointers and inheritance, the *polymorphism* feature of C++ allows the selection of a given module at the runtime.

This is the basic sequence underlying the automation of the PARTONS code, discussed below in Sect. 4.4. This works *mutatis mutandis* for all modules.

4.3 Services

The *Services* hide the complexity of low-level functions to provide high-level features to the users. A single service is basically a toolbox for the user: the user is given tools to use the software without knowing details of its operating.[2] The Services demonstrate their relevance in computations that combine several different objects. Before the inclusion of our GPD codes in the PARTONS framework, we had to take the outputs from various objects, like *e.g.* some GPD values, to manually run an evolution code and then feed the code computing CFFs. These operations were hand-made, with all the risks this implies. In the PARTONS structure, the Services combine different modules and data sets to produce results in a transparent way. Among others, `GPDService`

[1] It is not a copy of a pointer, which would still points to the same object. It is a *duplication* of the object, referred to by a new pointer.

[2] As an image, we can say that we can start a car by turning a key, and not knowing the detailed description of the motor and of electric circuits between the motor and the key.

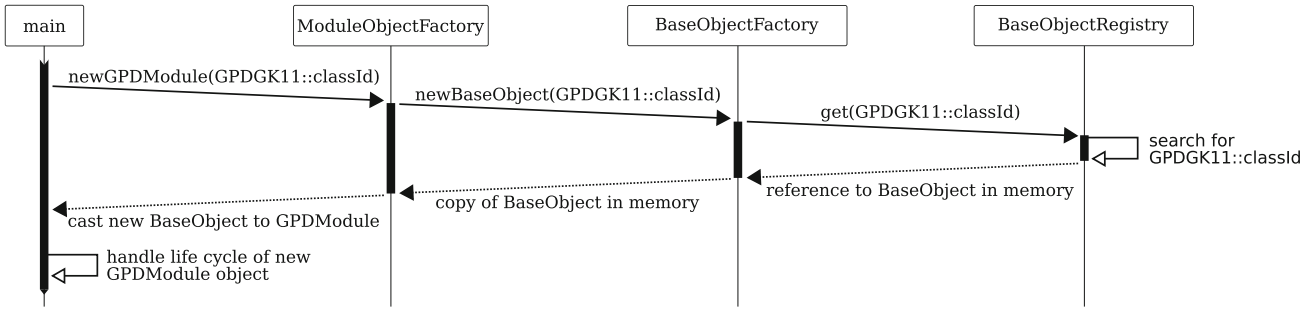


Fig. 7 Sequence diagram presenting the different processes, arranged in time, relating the different objects allowing the instantiation of a new module from the sole information of its identifier or name. See Fig. 6 for the description of diagram elements

provides several functions that hide the complexity related to repetitive tasks, like the evaluation of GPDs for a list of kinematic configurations. The following excerpt shows what are the presently offered operations:

```
class GPDService: public ServiceObject {

public:

    // Method used in automation to compute given tasks
    virtual void computeTask(Task &task);

    // Computes GPD model at specific kinematics
    // and for a provided list of GPD types
    GPDResult computeGPDModel(const GPDKinematic &gpdKinematic,
GPDModule* pGPDModule, const List<GPDType> & gpdType = List<GPDType
>()) const;

    // Computes GPD model for a list of kinematics
    // and for a provided list of GPD types
    // If needed, it can store the obtained results in a database
    List<GPDResult> computeManyKinematicOneModel(const List<
GPDKinematic> &gpdKinematicList, GPDModule* pGPDModule, const List<
GPDType> &gpdTypeList = List<GPDType>(), const bool storeInDB = 0);
};
```

Here, three different types of operations are available:

- The function `computeTask` is generic and is one of the building blocks of the automation in PARTONS.
- `computeGPDModel` evaluates all or only restricted types of GPDs for a single model and for a single kinematic configuration.
- `computeManyKinematicOneModel` evaluates all or only restricted types of GPDs for a single model and for a list of kinematic configurations. If needed, the obtained results can be stored in a database. The insertion is identified by a unique computation id returned to the standard output. The kinematic configurations can be bunched into a set of packets, with the size of each packet defined *via* the configuration file of PARTONS. In such a case, each packet can be evaluated in a separate thread, see Sect. 4.6.

Thanks to the services, repetitive tasks are coded and validated only once, which saves many possible implementation errors.

4.4 Automation and scenario manager

What the end-user really wants is just selecting the various models, kinematic configurations and observables to compute by specifying the necessary physical hypothesis. At the end of the day, this should be accomplished with a simple file, or through a web page. This would allow the PARTONS software to be used by physicists unfamiliar to C++, which represents a significant part of the theoretical physics community. We designed a functionality to run the code by sending the appropriate information, referred to as the *scenario*, through an XML file. This offers several advantages. First, the file can be read or manually written by a simple adaptation. Second, it can be easily generated by a web or graphical interface (such as the planned visualization tool). Third, the freedom in defining markups allows a structure very similar to that of the underlying C++ objects. For example, the computation of beam-spin asymmetry A_{LU}^{-} defined in Sect. 2.2.3 reads:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<!-- Scenario starts here -->
<!-- For your convenience and for the bookkeeping you can provide creation date
and a unique description -->
<scenario date="2017-07-18" description="DVCS observable
evaluation for single kinematics example">

<!-- First task: evaluate DVCS observable for a single kinematics -->
<!-- Indicate service and its methods to be used and indicate if the result should be
stored in the database -->
<task service="ObservableService" method="computeObservable"
storeInDB="0">

 <!-- Define DVCS observable kinematics -->
 <kinematics type="ObservableKinematic">
  <param name="xB" value="0.2" />
  <param name="t" value="-0.1" />
  <param name="Q2" value="2." />
  <param name="E" value="6." />
 </kinematics>

 <!-- Define all physics assumptions -->
 <computation_configuration>

  <!-- Select DVCS observable -->
  <module type="Observable" name="DVCSAluMinus">

   <!-- Select DVCS process model -->
   <module type="ProcessModule" name="DVCSProcessGV08">

    <!-- Select xi-converter module -->
```

```
29    <!-- (it is used to evaluate GPD variable xi out of kinematics) -->
30    <module type="XiConverterModule" name="XiConverterXBToXi">
31    </module>
32
33    <!-- Select scales module -->
34    <!-- (it is used to evaluate factorization and renormalization scales out of
      kinematics) -->
35    <module type="ScalesModule" name="ScalesQ2Multiplier">
36
37     <!-- Configure this module -->
38     <param name="lambda" value="1." />
39    </module>
40
41    <!-- Select DVCS CFF model -->
42    <module type="ConvolCoeffFunctionModule" name="
      DVCSCFFStandard">
43
44     <!-- Indicate pQCD order of calculation -->
45     <param name="qcd_order_type" value="LO" />
46
47     <!-- Select GPD model -->
48     <module type="GPDModule" name="GPDGK11">
49     </module>
50
51    </module>
52   </module>
53  </module>
54 </computation_configuration>
55 </task>
56
57 <!-- Second task: print results of the last computation into standard output -->
58 <task service="ObservableService" method="printResults">
59 </task>
60
61 </scenario>
```

This XML file is parsed by an object named `ScenarioManager`, which now possesses a collection of `string` objects. The real difficulty is the creation of C++ objects from this collection of `string` objects. All scenarios start with the specification of target service and method inside that service, which processes all `string` objects enclosed between the two markups `<task>` and `</task>`. The involved service "knows" what the method being selected really needs to perform the computation, and looks through the lists of objects and parameters, if all the relevant information is provided. Each service possesses a `computeTask` method, and only services do have such methods. The role of the `computeTask` functions is the distribution of the `ParameterList` objects to the different constructors of the various objects required to perform the considered task. Centralizing the creation of all the objects in the `computeTask` function of a service gives robustness to the generic use of an XML scenario. Then, for the beans (inputs and outputs), everything is taken care of by the constructors. For modules, the code calls the Factory to get the required objects by their names. Namely, the Factory gets the name, checks in the Registry[3] if there exists a pointer corresponding to that name, and either gives a copy of the object, or sends an error message. Each module is then configured from the list of parameters associated to the module name. At last, the service gives all objects that have just been constructed as parameters of the target function (`computeObservable` here).

The whole sequence is recapitulated in Fig. 8. The XML file dictates the evaluation of the selected DVCS observable at a kinematic configuration (x_B, t, Q^2, E, ϕ), where E is the beam energy in the LAB system. In our example, the observable is $A_{\mathrm{LU}}^{-}(\phi)$, which is the ratio of combinations of DVCS cross sections. The kinematic configuration (x_B, t, Q^2, E, ϕ) is then transferred to the chosen class inherited from `DVCSModule`, say `DVCSProcessGV08`. To evaluate cross sections, the parent class requires some values of CFFs, so it turns to `DVCSConvolCoeffFunctionModule` with the kinematics $(\xi, t, Q^2, \mu_F^2, \mu_R^2)$. The values of ξ and (μ_F^2, μ_R^2) are evaluated from the input kinematic configuration by `XiConverterModule` and `ScalesModule` modules, respectively. In our example we have $\xi = x_B/(2 - x_B)$ and $\mu_R^2 = \mu_F^2 = \lambda Q^2$ with $\lambda = 1$, but other possibilities exist. The evaluation of CFFs means computing integrals and hence probing GPDs (here `GPDGK11`) over x at $(x, \xi, t, \mu_F^2, \mu_R^2)$ with x *a priori* selected by the integration routine, and renormalization and factorization scales chosen by the user as part of his modeling assumptions. The kinematic configuration at which the observable is evaluated has been converted and transmitted from top to bottom. The other way around returns sequentially the evaluation of the GPDs, of the CFFs, of the cross section and of the considered asymmetry. The final result can be stored in the used database (`storeInDB` switch).

The automation file is totally generic: it is independent of the different modules or services. We can use a generic parser and a generic description of XML files. It is one more answer to our need of flexibility. The architecture of PARTONS can evolve without any modifications in the parser or in the XML file description.

4.5 Database: storage and transactions

A database is needed for several reasons. We want to keep track of the results of our computations, and once a result is validated to keep it and not compute it again anymore. With a related database entry containing the XML file producing the result, it becomes easy to see how something was computed, even if we ask ourselves a long time after. It may also well be that the computational cost of some GPD model is prohibitive, in which case the predictions of this model can be computed separately on a dedicated cluster, stored in the database and then used for the computation of an observable. The structure of `GPDModule` is designed so as to make transparent to the user the fact that the GPD values come from a database instead of a direct numerical evaluation. At last, we can also store experimental results in the database, and make systematic comparisons of experimental results and theoretical predictions. A database is optimized to make

[3] The Registry contains a dictionary that associates a `classId` to a name given as a `string` variable.

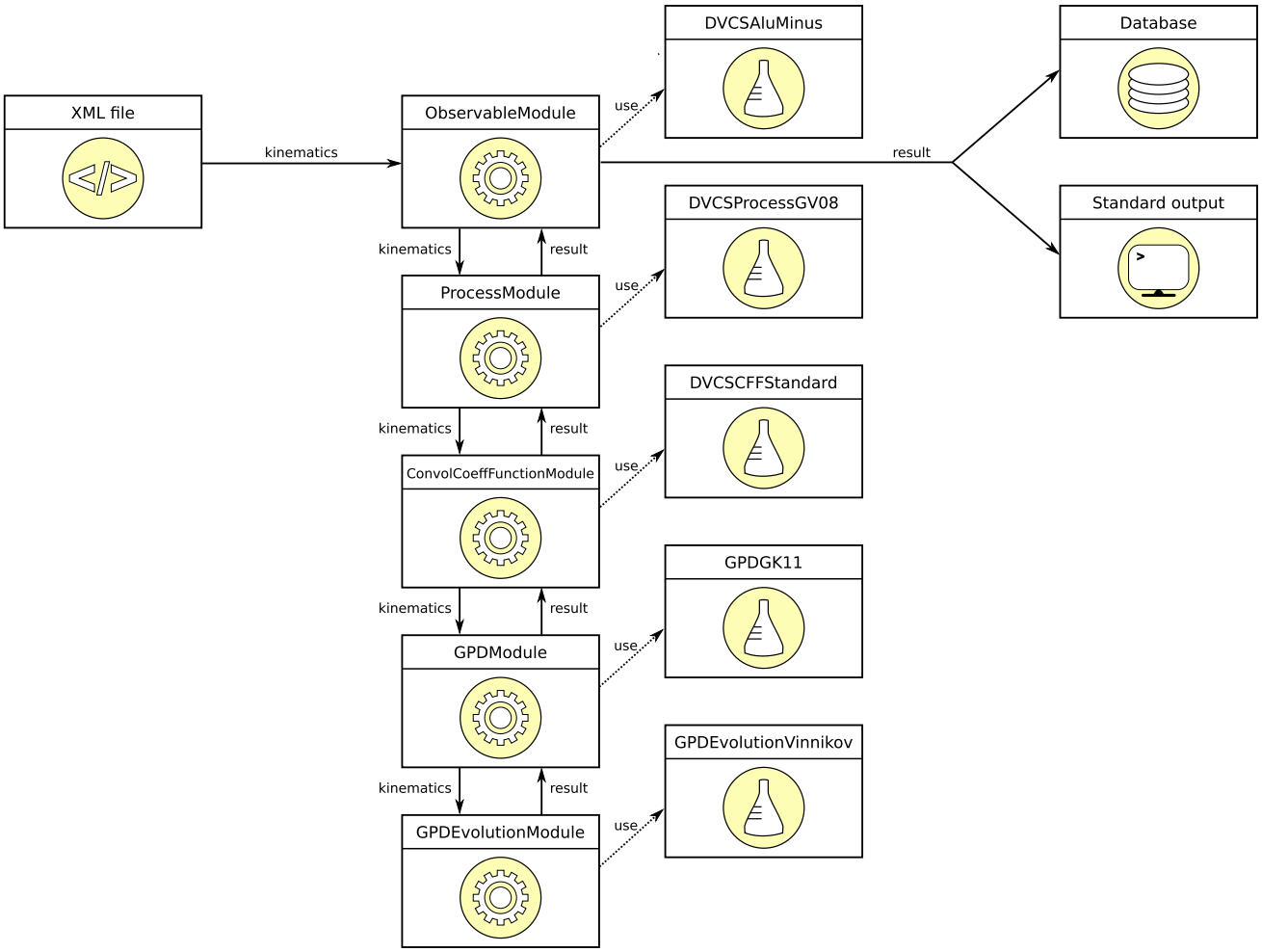


Fig. 8 Principle of automation: computation of a beam spin asymmetry

data selections, in which case kinematic cuts (for example a cut on $-t/Q^2$ to probe the Bjorken limit) are simple and efficient.

In the PARTONS architecture, one layer is dedicated to the transactions with databases, *cf.* Sect. 4.1 and Fig. 4. At this level, we should ignore what is the explicit type of database (*e.g.* a local database on a laptop, or a database on a distant server).

The PARTONS code manages the transactions by using *Data Access Objects* (DAOs) and the related services. For the sake of simplicity, we discuss the case of a single computation of GPD value. We store both the GPD value and the associated kinematics. Thus, there are two DAO services involved, `GPDKinematicDaoService` and `GPDResultDaoService`, which transform the corresponding C++ objects into a collection of simple types (`int`, `double`, `string`, *etc.*) - it is the *serialization* step. The DAO services obey the same pattern as the other services in PARTONS: they receive as inputs and return as results high-level objects instead of simple types. The DAO services then call the necessary DAO objects.

In our example, there are two of them: `GPDKinematicDao` and `GPDResultDao`. In a DAO object we can define as many functions as there are requests to make to the database, *e.g.* in the case of `GPDKinematicDao`: `insert` (to add a kinematic configuration in the database), `select` (to read a kinematic configuration in the database), `delete` (to suppress a kinematic configuration in the database), *etc.* Any type of requests can be implemented that way. The following excerpt shows the code underlying the insertion in the database, by the `GPDKinematicDao` object, of a single GPD kinematic configuration:

```
1 int GPDKinematicDao::insert(double x, double xi, double t, double MuF2, double
  MuR2) const {
2
3   // Returned value (last inserted id if everything fine)
4   int result = -1;
5
6   // Initialize QSqlQuery object
7   QSqlQuery query(DatabaseManager::getInstance()->getProductionDatabase());
8
```

```
    // Prepare the query
    ElemUtils::Formatter formatter;
    formatter << "INSERT INTO " << Database::TABLE_NAME_GPD_KINEMATIC
              << " (x, xi, t, MuF2, MuR2) VALUES (:x, :xi, :t, :MuF2, :MuR2)";

    query.prepare(QString(formatter.str().c_str()));

    // Bind values
    query.bindValue(":x", x);
    query.bindValue(":xi", xi);
    query.bindValue(":t", t);
    query.bindValue(":MuF2", MuF2);
    query.bindValue(":MuR2", MuR2);

    // Execute query, look for errors
    if (query.exec()) {
      result = query.lastInsertId().toInt();
    } else {
      throw ElemUtils::CustomException(getClassName(), __func__,
            ElemUtils::Formatter() << query.lastError().text().toStdString()
            << " for sql query = "
            << query.executedQuery().toStdString());
    }

    // Return
    return result;
}
```

PARTONS generates a SQL-like request, which is a simple `string`. This text is interpreted by `Qt` to replace the dynamical fields (here x, ξ, t, μ_F^2 and μ_R^2) by their actual values (here `double` variables). The `Qt` management of connectors[4] makes possible to send the same SQL request to databases of different types, like `MySQL` [102] and `SQLite` [103]. The connection of the PARTONS objects, specific to our needs, to the simple types in the database, is done once, and only once, whatever the type of the database is.

The `GPDKinematicDaoService` performs the same tasks, but this time with a single or a list of `GPDKinematic` objects. By default, PARTONS uses a transaction mechanism, which allows to "rollback" all modifications done during a single insertion session. This prevents in particular from a disintegration of the database content in a case of failed transaction. The way it is achieved is illustrated by the following code:

```
int GPDKinematicDaoService::insert(const List<GPDKinematic>& gpdKinematicList) const {

    // Returned value (last inserted id if everything fine)
    int gpdKinematicId = -1;

    // Indicate transaction mechanism
    QSqlDatabase::database().transaction();

    try {

      //Add each kinematic object
      for (unsigned int i = 0; i != gpdKinematicList.size(); i++) {
        gpdKinematicId = insertWithoutTransaction(gpdKinematicList.get(i));
      }

      // If there is no exception we can commit all queries
      QSqlDatabase::database().commit();

    } catch (const std::exception &e) {

      // In a case of problems revert changes
      // i.e. put database to the previous (stable) state
      QSqlDatabase::database().rollback();

      // Indicate error
      throw ElemUtils::CustomException(getClassName(), __func__, e.what());
    }

    // Return
    return gpdKinematicId;
}
```

where `insertWithoutTransaction` unfolds `GPDKinematic` into simple types and runs the `insert` function of `GPDKinematicDao`:

```
int GPDKinematicDaoService::insertWithoutTransaction(const GPDKinematic& gpdKinematic) const {
  return m_GPDKinematicDao.insert(gpdKinematic.getX(), gpdKinematic.getXi(),
          gpdKinematic.getT(), gpdKinematic.getMuF2(), gpdKinematic.getMuR2());
}
```

Note, that there is as many DAO classes as there are tables in the database. In that respect, the database structure reflects the architecture of the C++ code.

4.6 Threads

Threads are sequences of instructions that can be independently managed by the user's operating system. On multiprocessor/multicore computers (basically all modern machines), threads can be executed simultaneously, in contrary to an execution of only one process at a time. This allows to exploit the full capacity of current computers (including those available at computing farms), which allows a significant reduction of the computation time. In PARTONS, threads are used exclusively by Services, but one thread is also reserved for the Logger, which processes human-understandable messages streamed from the code during its execution. The threads are managed by using the `SFML` library. In particular, this library is used to protect sensitive areas of the allocated memory from being modified by several threads at the same time.

4.6.1 Threads and services

The PARTONS framework offers a possibility to use threads in Services, whenever one needs to make an evaluation for a large set of kinematic configurations. The mechanism is fully automated, so the use of threads is implicit for the users. We will illustrate how threads are used by the Services on an exemplary calculation of many GPD kinematic configurations. From the user's point of view, whenever one wants to use threads, one needs only to run the appropriate method of `GPDService`, that is `computeManyKinematicOneModel` that has been already described in Sect. 4.3:

```
List<GPDResult> computeManyKinematicOneModel(const
List<GPDKinematic> &gpdKinematicList, GPDModule* pGPDModule, const
List<GPDType> &gpdTypeList = List<GPDType>(), const bool storeInDB =
0);
```

[4] A connector is the library provided by the editors of the database which permits transactions with a database. This library is written in different languages, *e.g.* C++, Java, Python, …

In our example, `gpdKinematicList` contains many kinematic configurations to be evaluated by PARTONS. Note, that `computeManyKinematicOneModel` can be executed by the users explicitly if they code in C++, or implicitly in the automation if they process an input XML file. The key of multithreading are two parameters set in the configuration file of PARTONS. The first of them is the number of available processor cores, defining the number of threads that can run in parallel. The second parameter defines how many kinematic configurations should be evaluated in a single thread. The involved Service supervises then the use of threads automatically: it divides kinematic configurations into separate packets, runs each of such packets in a separate thread, waits until all packets are evaluated, and finally returns results as they were evaluated in a single process.

4.6.2 Logger

On top of the thread(s) dedicated to computations, PARTONS uses an additional thread serving as the *Logger*. The code sends information at four different levels: DEBUG, INFO, WARNING and ERROR. With the first three levels the code is always running, while the ERROR level forces the code to stop. Sending the information to the Logger does not slow down the computations by taking precious time to screen (or file) printing. Warning messages signal to the user that there is something to be checked carefully, *e.g.* slow convergence in a numerical routine. The output of the Logger can be directed either to the screen, or to a log file, or to both of them. It traces back all details of the computation that have been considered relevant by the developer.

5 Existing modules

So far, only the DVCS process is implemented in the framework. However, PARTONS was designed with the idea of an easy addition of other partonic processes, not necessary limited to GPDs. Beyond DVCS, in the near future, TCS and HEMP will be integrated into PARTONS, both important from the point of view of existing and foreseen measurements. The following modules are currently integrated in the PARTONS framework:

GPD	The GK model [97–99], the VGG model [7,66,100,101], and the GPD models used in the papers by Vinnikov [104], Moutarde et al. [63] and Mezrag et al. [105].
Evolution	The Vinnikov code [104].
CFF	The LO and NLO evaluation used in Ref. [63] together with its extension to the massive quark case [106], and the LO evaluation used in Refs. [7,66,100,101].
DVCS	The set of unpublished analytic expressions of Guichon and Vanderhaeghen used in Refs. [75,107], and the latest set of expressions [53] by Belitsky and Müller.
Alpha	Four-loop perturbative running of the strong coupling constant from PDG [108], and constant value.

In a nutshell, the present version of PARTONS contains all the necessary tools to perform DVCS studies at leading twist and NLO. The presented set of modules is by no means limiting. Other modules will be integrated in the framework to allow systematic differential studies requiring the flexible design of PARTONS. All the previous categories should be extended by new modules, either to recover the features of existing codes, or to test brand new development in the integrated chain between models and measurements. For instance, subleading twist contributions or transversity GPDs can be studied within the framework by adding or extending the previous modules. In principle, these are already dealt with at the level of the DVCS cross section but are considered vanishing given that the existing GPD models and CFF modules disregard them.

6 Examples

The source code of PARTONS and the pre-configured appliances of the provided virtual machines can be downloaded for the project web page: `http://partons.cea.fr`. Two kinds of virtual machines are available: the light version with only the runtime environment aimed at the users, and the developer version containing both the runtime and development environments.

The web page serves also as a main source of the technical information on PARTONS. On top of a detailed description of the code elements, the users may also find there many useful tutorials, such as a quick guide helping them to start their experience with PARTONS, an additional help with installation technicalities, tips on using our virtual machines, templates for adding new modules and many more. Examples on how to use specific elements of the PARTONS framework are available online, but they are also provided as one of the sub-projects called `partons-example`. Some of those examples are shown here to demonstrate the proper handling of PARTONS and its capabilities.

6.1 Structure of main function

PARTONS as a library can be used in any C++ code. However, one should always remember that PARTONS requires a proper initialization and handling of the exceptions during the execution. In general, this should be encoded in the

`main` function of the executable. An example is provided here, where we also show to correctly process input XML files in the automation.

```
#include <ElementaryUtils/logger/CustomException.h>
#include <ElementaryUtils/logger/LoggerManager.h>
#include <partons/Partons.h>
#include <partons/services/automation/AutomationService.h>
#include <partons/ServiceObjectRegistry.h>
#include <QtCore/qcoreapplication.h>
#include <string>
#include <vector>

int main(int argc, char** argv) {

    // Init Qt4
    QCoreApplication a(argc, argv);
    PARTONS::Partons* pPartons = 0;

    try {

        // Init PARTONS application
        pPartons = PARTONS::Partons::getInstance();
        pPartons->init(argc, argv);

        // RUN XML SCENARIO

        // You need to provide at least one scenario
        // via executable argument
        if (argc <= 1) {

            throw ElemUtils::CustomException("main", __func__,
                "Missing argument, please provide one or more
than one XML scenario file.");
        }

        // Get arguments to retrieve xml file path list.
        std::vector<std::string> xmlScenarioFilePathList(argc - 1);

        for (unsigned int i = 1; i < argc; i++) {
            xmlScenarioFilePathList[i - 1] = argv[i];
        }

        // Retrieve automation service parse scenario xml file
        // and play it.
        PARTONS::AutomationService* pAutomationService =
            pPartons->getServiceObjectRegistry()->getAutomationService();

        for (unsigned int i = 0; i < xmlScenarioFilePathList.size(); i++) {
            PARTONS::Scenario* pScenario = pAutomationService->parseXMLFile(
                xmlScenarioFilePathList[i]);
            pAutomationService->playScenario(pScenario);
        }

        // RUN CPP CODE

        // You can put your own code here and build
        // a stand-alone program based on PARTONS library.

        // computeSingleKinematicsForGPD();
    }
    // Appropriate catching of exceptions is crucial
    // for working of PARTONS. It defines its own type of
    // exception, which allows to display class name and
    // function name where the exception has occurred,
    // but also a human readable explanation.
    catch (const ElemUtils::CustomException &e) {

        // Display what happened
        pPartons->getLoggerManager()->error(e);

        // Close PARTONS application properly
        if (pPartons) {
            pPartons->close();
        }
    }
    // In a case of standard exception.
    catch (const std::exception &e) {

        // Display what happened
        pPartons->getLoggerManager()->error("main", __func__, e.what());

        // Close PARTONS application properly
        if (pPartons) {
            pPartons->close();
        }
    }

    // Close PARTONS application properly
    if (pPartons) {
        pPartons->close();
    }

    return 0;
}
```

6.2 Computation of GPD for a single kinematic configuration without automation

The following example shows how to compute GPDs for a single kinematic configuration without the automation. Almost each line of the code corresponds to a physical hypothesis, but the user still has to explicitly deal with the pointers. The function can be executed in `main` in the specified place of the previous excerpt.

```
#include <ElementaryUtils/logger/LoggerManager.h>
#include <partons/beans/gpd/GPDKinematic.h>
#include <partons/modules/gpd/GPDMMS13.h>
#include <partons/ModuleObjectFactory.h>
#include <partons/Partons.h>
#include <partons/services/GPDService.h>
#include <partons/ServiceObjectRegistry.h>

void computeSingleKinematicsForGPD() {

    // Retrieve GPD service
    PARTONS::GPDService* pGPDService =
        PARTONS::Partons::getInstance()->getServiceObjectRegistry()->
getGPDService();

    // Create GPD module with the BaseModuleFactory
    PARTONS::GPDModule* pGPDModel =
        PARTONS::Partons::getInstance()->getModuleObjectFactory()->
newGPDModule(
            PARTONS::GPDMMS13::classId);

    // Create a GPDKinematic(x, xi, t, MuF2, MuR2) to compute
    PARTONS::GPDKinematic gpdKinematic(0.1, 0.2, -0.1, 2., 2.);

    // Run computation
    PARTONS::GPDResult gpdResult = pGPDService->computeGPDModel(
gpdKinematic,
        pGPDModel);

    // Print results
    PARTONS::Partons::getInstance()->getLoggerManager()->info("main",
__func__,
        gpdResult.toString());

    // Remove pointer references
    // Module pointers are managed by PARTONS
    PARTONS::Partons::getInstance()->getModuleObjectFactory()->
updateModulePointerReference(
        pGPDModel, 0);
    pGPDModel = 0;
}
```

6.3 Computation of GPD for single kinematic configuration with automation

The following example shows how to compute GPDs for a single kinematic configuration with the automation. Each line of the XML file corresponds to a physical hypothesis,

so the user does not have to explicitly deal with the pointers anymore.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<!-- Scenario starts here -->
<!-- For your convenience and for bookkeeping provide creation date and unique description -->
<scenario date="2017-07-18" description="GPD evaluation for single kinematics example">

  <!-- First task: evaluate GPD model for a single kinematics -->
  <!-- Indicate service and its methods to be used and indicate if the result should be stored in the database -->
  <task service="GPDService" method="computeGPDModel" storeInDB="0">

    <!-- Define GPD kinematics -->
    <kinematics type="GPDKinematic">
      <param name="x" value="0.1" />
      <param name="xi" value="0.2" />
      <param name="t" value="-0.1" />
      <param name="MuF2" value="2." />
      <param name="MuR2" value="2." />
    </kinematics>

    <!-- Define physics assumptions -->
    <computation_configuration>

      <!-- Select GPD model -->
      <module type="GPDModule" name="GPDMMS13">
      </module>

    </computation_configuration>

  </task>

  <!-- Second task: print results of the last computation into standard output -->
  <task service="GPDService" method="printResults">
  </task>

</scenario>
```

6.4 Computation of beam spin asymmetry for many kinematic configurations with automation

The following example shows how to compute the beam spin asymmetry $A_{LU}^{-}(\phi)$ defined in Eq. (23) for a set of kinematic configurations typical to Jefferson Lab upgraded to 12 GeV: $x_B = 1/3$ ($\xi \simeq 0.2$), $t = -0.2$ GeV^2, $Q^2 = 4$ GeV^2 and beam energy $E_{\text{Lab}} = 11$ GeV. This example is close to the one discussed in Sect. 4.4, but this time the code is executed with a list of values of ϕ ranging between 0 and 360 degrees. This is indicated by the method `computeManyKinematicOneModel`. The list of kinematic configurations is provided in a file as indicated between the markups `<ObservableKinematic>` and `</ObservableKinematic>`, and is described as simple text:

```
0.333|-0.2|4.0|11.|0.0
0.333|-0.2|4.0|11.|-3.6
0.333|-0.2|4.0|11.|-7.2
0.333|-0.2|4.0|11.|-10.8
0.333|-0.2|4.0|11.|-14.4
 ...
```

where we can see, on each line, from left to right: x_B, t (in GeV^2), Q^2 (in GeV^2), E_{Lab} (in GeV) and ϕ (in degrees).

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<!-- Scenario starts here -->
<!-- For your convenience and for the bookkeeping you can provide creation date and a unique description -->
<scenario date="2017-07-18" description="DVCS observable evaluation for many kinematics example">

<!-- First task: evaluate DVCS observable for a single kinematics -->
<!-- Indicate service and its methods to be used and indicate if the result should be stored in the database -->
<task service="ObservableService" method="computeManyKinematicOneModel" storeInDB="1">

 <!-- Define DVCS observable kinematics -->
 <kinematics type="ObservableKinematic">

  <!-- Path to file defining kinematics -->
  <param name="file" value="kinematics_dvcs_observable.csv" />
 </kinematics>

 <!-- Define all physics assumptions -->
 <computation_configuration>

  <!-- Select DVCS observable -->
  <module type="Observable" name="DVCSAluMinus">

   <!-- Select DVCS process model -->
   <module type="ProcessModule" name="DVCSProcessGV08">

    <!-- Select xi-converter module -->
    <!-- (it is used to evaluate GPD variable xi out of kinematics) -->
    <module type="XiConverterModule" name="XiConverterXBToXi">
    </module>

    <!-- Select scales module -->
    <!-- (it is used to evaluate factorization and renormalization scales out of kinematics) -->
    <module type="ScalesModule" name="ScalesQ2Multiplier">

     <!-- Configure this module -->
     <param name="lambda" value="1." />
    </module>

    <!-- Select DVCS CFF model -->
    <module type="ConvolCoeffFunctionModule" name="DVCSCFFStandard">

     <!-- Indicate pQCD order of calculation -->
     <param name="qcd_order_type" value="LO" />

     <!-- Select GPD model -->
     <module type="GPDModule" name="GPDGK11">
     </module>

    </module>
   </module>
  </module>
 </computation_configuration>
</task>

</scenario>
```

In this example the obtained values are stored in the database (`storeInDB` switch set to `1`). After the successful insertion the code returns such a line to the Logger (which itself outputs to the standard output and/or a file):

```
[INFO] (ObservableService::computeTask)
       ObservableResultList object
has been stored in database with
       computation_id = 1
```

The inserted data can be then fetched from the database by running such a scenario:

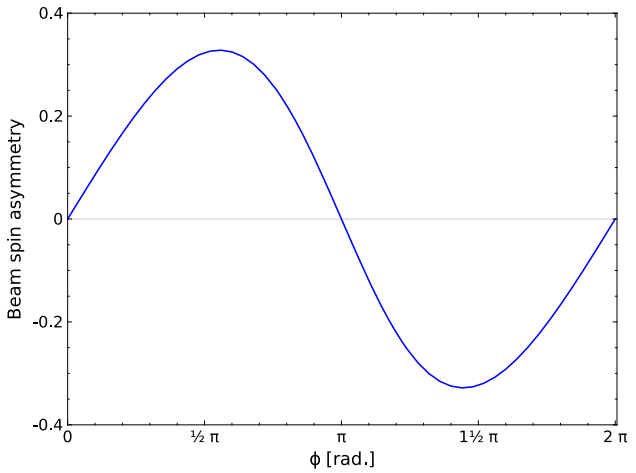


Fig. 9 Beam spin asymmetry $A_{\mathrm{LU}}^{-}(\phi)$ for $x_B = 1/3$, $t = -0.2$ GeV^2, $Q^2 = 4$ GeV^2, and $E_{\mathrm{Lab}} = 11$ GeV. Compton Form Factors are evaluated at LO approximation with the GK GPD model [97–99]

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>

<!-- Scenario starts here -->
<!-- For your convenience and for bookkeeping provide creation date and unique
description -->
<scenario date="2017-07-18" description="Get observable result
from database">

 <!-- Task: generate file with data matching indicated criteria -->
 <task service="ObservableService" method="generatePlotFile">

  <!-- Variables selected to be stored in the output file -->
  <task_param type="select">
   <param name="xPlot" value="phi" />
   <param name="yPlot" value="observable_value" />
  </task_param>

  <!-- Applied requirements -->
  <task_param type="where">
   <param name="computation_id" value="1" />
  </task_param>

  <!-- Path to the output file -->
  <task_param type="output">
   <param name="filePath" value="output.dat" />
  </task_param>
 </task>
</scenario>
```

where the value specified in `<param name="computation_id" />` tag is specific to the initial computation. As a result, a text file is created (here: `output.dat`) containing pre-formated values that one can use for instance to make a plot, like the one shown in Fig. 9.

7 Licenses

One of the goals of the developers team is to allow PARTONS to spread through the community and for that purpose, we have adopted a license prescription in such a way that users are free to use and modify the source code of PARTONS and all its dependencies. More precisely, PARTONS is distributed under the GPLv3 license, and so are the sub-projects `numa` and `partons-examples`. The sub-project `elementary-utils` is distributed under the Apache license. Users are encouraged to contribute back their own developments by joining our forge, a `GitLab` server with a `Git` repository: https://drf-gitlab.cea.fr/partons/core

Concerning our external libraries, the `SFML` library is available under the `zlib/png` license while `CLN` and `Qt` are under `GPLv3`.

8 Conclusions

In the last twenty years we have witnessed an intense theoretical and experimental activity in the field of exclusive processes described in terms of Generalized Parton Distributions. It is also a crucial part of the forthcoming experiments at Jefferson Lab, COMPASS and in the future at EIC or LHeC. The amount and quality of the expected data, together with the richness and versatility of theoretical approaches to its description, calls for a flexible, stable and accurate software framework that will allow for systematic phenomenological studies. In this paper we have described such a framework, called PARTONS, and how it addresses the most important tasks: automation, modularity, non-regression and data storage. We have also provided examples of simple XML *scenario* files illustrating automated calculations of physical observables.

Since its inception the PARTONS framework has expanded rapidly with the addition of new core developers. We intend, in the mid- to long-term future, to complement PARTONS with new theoretical developments, new computing techniques and other exclusive processes. To achieve this, we expect that more physicists will join the development team to integrate new modules and benefit from our integrated chain, relating theory to experimental observables. PARTONS should become the *de facto* software framework for the GPD analysis of the next-generation exclusive data.

Acknowledgements The authors would like to thank P. Aguilera, S. Anvar, A. Besse, D. Chapon, R. Géraud, P. Guichon, K. Joo, A. Kielb, K. Kumerički, K. Passek-Kumerički and J.-Ph. Poli for many fruitful discussions and valuable inputs. This work was supported in part by the Commissariat à l'Energie Atomique et aux Energies Alternatives, by the French National Research Agency (ANR) grant ANR-12-MONU-0008-01, by the Grant No. 2017/26/M/ST2/01074 of the National Science Centre, Poland, and by the DOE grant DE-FG-04-ER41309.

References

1. D. Mueller, D. Robaschik, B. Geyer, F.M. Dittes, J. Hoeji, Fortschr. Phys. **42**, 101 (1994). https://doi.org/10.1002/prop.2190420202
2. A.V. Radyushkin, Phys. Lett. B **380**, 417 (1996). https://doi.org/10.1016/0370-2693(96)00528-X
3. X.D. Ji, Phys. Rev. D **55**, 7114 (1997). https://doi.org/10.1103/PhysRevD.55.7114
4. A. Airapetian et al., Phys. Rev. Lett. **87**, 182001 (2001). https://doi.org/10.1103/PhysRevLett.87.182001
5. S. Stepanyan et al., Phys. Rev. Lett. **87**, 182002 (2001). https://doi.org/10.1103/PhysRevLett.87.182002
6. X.D. Ji, J. Phys. G **24**, 1181 (1998). https://doi.org/10.1088/0954-3899/24/7/002
7. K. Goeke, M.V. Polyakov, M. Vanderhaeghen, Progr. Part. Nucl. Phys. **47**, 401 (2001). https://doi.org/10.1016/S0146-6410(01)00158-2
8. M. Diehl, Phys. Rep. **388**, 41 (2003). https://doi.org/10.1016/j.physrep.2003.08.002
9. A.V. Belitsky, A.V. Radyushkin, Phys. Rep. **418**, 1 (2005). https://doi.org/10.1016/j.physrep.2005.06.002
10. S. Boffi, B. Pasquini, Riv. Nuovo Cim. **30**, 387 (2007). https://doi.org/10.1393/ncr/i2007-10025-7
11. M. Guidal, H. Moutarde, M. Vanderhaeghen, Rep. Progr. Phys. **76**, 066202 (2013). https://doi.org/10.1088/0034-4885/76/6/066202
12. D. Mueller, Few Body Syst. **55**, 317 (2014). https://doi.org/10.1007/s00601-014-0894-3
13. N. d'Hose, S. Niccolai, A. Rostomyan, Eur. Phys. J. A **52**(6), 151 (2016). https://doi.org/10.1140/epja/i2016-16151-9
14. K. Kumericki, S. Liuti, H. Moutarde, Eur. Phys. J. A **52**(6), 157 (2016). https://doi.org/10.1140/epja/i2016-16157-3
15. J. Dudek et al., Eur. Phys. J. A **48**, 187 (2012). https://doi.org/10.1140/epja/i2012-12187-1
16. F. Gautheron, et al., COMPASS-II Proposal. Tech. Rep. CERN-SPSC-2010-014. SPSC-P-340, CERN, Geneva (2010). URL https://cds.cern.ch/record/1265628
17. A. Accardi et al., Eur. Phys. J. A **52**(9), 268 (2016). https://doi.org/10.1140/epja/i2016-16268-9
18. J.L.A. Fernandez et al., J. Phys. **G39**, 075001 (2012). https://doi.org/10.1088/0954-3899/39/7/075001
19. E.R. Berger, M. Diehl, B. Pire, Eur. Phys. J. C **23**, 675 (2002). https://doi.org/10.1007/s100520200917
20. L. Favart, M. Guidal, T. Horn, P. Kroll, Eur. Phys. J. A **52**(6), 158 (2016). https://doi.org/10.1140/epja/i2016-16158-2
21. A.V. Belitsky, D. Mueller, Phys. Rev. Lett. **90**, 022001 (2003). https://doi.org/10.1103/PhysRevLett.90.022001
22. M. Guidal, M. Vanderhaeghen, Phys. Rev. Lett. **90**, 012001 (2003). https://doi.org/10.1103/PhysRevLett.90.012001
23. DYu. Ivanov, A. Schafer, L. Szymanowski, G. Krasnikov, Eur. Phys. J. C **34**(3), 297 (2004). https://doi.org/10.1140/epjc/s2004-01712-x,10.1140/epjc/s10052-015-3298-8. ([Erratum: Eur. Phys. J. C75, no.2,75(2015)])
24. R. Boussarie, B. Pire, L. Szymanowski, S. Wallon, JHEP **02**, 054 (2017). https://doi.org/10.1007/JHEP02(2017)054
25. A. Pedrak, B. Pire, L. Szymanowski, J. Wagner, Phys. Rev. D **96**(7), 074008 (2017). https://doi.org/10.1103/PhysRevD.96.074008
26. B.Z. Kopeliovich, I. Schmidt, M. Siddikov, Phys. Rev. D **86**, 113018 (2012). https://doi.org/10.1103/PhysRevD.86.113018
27. B. Pire, L. Szymanowski, J. Wagner, Phys. Rev. D **95**(9), 094001 (2017). https://doi.org/10.1103/PhysRevD.95.094001
28. B. Pire, L. Szymanowski, J. Wagner, Phys. Rev. D **95**(11), 114029 (2017). https://doi.org/10.1103/PhysRevD.95.114029
29. S. Chekanov et al., Phys. Lett. B **573**, 46 (2003). https://doi.org/10.1016/j.physletb.2003.08.048
30. A. Aktas et al., Eur. Phys. J. C **44**, 1 (2005). https://doi.org/10.1140/epjc/s2005-02345-3
31. S. Chen et al., Phys. Rev. Lett. **97**, 072002 (2006). https://doi.org/10.1103/PhysRevLett.97.072002
32. A. Airapetian et al., Phys. Rev. D **75**, 011103 (2007). https://doi.org/10.1103/PhysRevD.75.011103
33. C.M. Camacho et al., Phys. Rev. Lett. **97**, 262002 (2006). https://doi.org/10.1103/PhysRevLett.97.262002
34. M. Mazouz et al., Phys. Rev. Lett. **99**, 242501 (2007). https://doi.org/10.1103/PhysRevLett.99.242501
35. F.D. Aaron et al., Phys. Lett. B **659**, 796 (2008). https://doi.org/10.1016/j.physletb.2007.11.093
36. F.X. Girod et al., Phys. Rev. Lett. **100**, 162002 (2008). https://doi.org/10.1103/PhysRevLett.100.162002
37. A. Airapetian et al., JHEP **06**, 066 (2008). https://doi.org/10.1088/1126-6708/2008/06/066
38. S. Chekanov et al., JHEP **05**, 108 (2009). https://doi.org/10.1088/1126-6708/2009/05/108
39. G. Gavalian et al., Phys. Rev. C **80**, 035206 (2009). https://doi.org/10.1103/PhysRevC.80.035206
40. F.D. Aaron et al., Phys. Lett. B **681**, 391 (2009). https://doi.org/10.1016/j.physletb.2009.10.035
41. A. Airapetian et al., JHEP **11**, 083 (2009). https://doi.org/10.1088/1126-6708/2009/11/083
42. A. Airapetian et al., Nucl. Phys. B **842**, 265 (2011). https://doi.org/10.1016/j.nuclphysb.2010.09.010
43. A. Airapetian et al., JHEP **06**, 019 (2010). https://doi.org/10.1007/JHEP06(2010)019
44. A. Airapetian et al., Phys. Lett. B **704**, 15 (2011). https://doi.org/10.1016/j.physletb.2011.08.067
45. A. Airapetian et al., JHEP **07**, 032 (2012). https://doi.org/10.1007/JHEP07(2012)032
46. S. Pisano et al., Phys. Rev. D **91**(5), 052014 (2015). https://doi.org/10.1103/PhysRevD.91.052014
47. H.S. Jo et al., Phys. Rev. Lett. **115**(21), 212003 (2015). https://doi.org/10.1103/PhysRevLett.115.212003
48. M. Defurne et al., Phys. Rev. C **92**(5), 055202 (2015). https://doi.org/10.1103/PhysRevC.92.055202
49. M. Defurne, et al., (2017)
50. A.V. Belitsky, D. Mueller, A. Kirchner, Nucl. Phys. B **629**, 323 (2002). https://doi.org/10.1016/S0550-3213(02)00144-X
51. A.V. Belitsky, D. Mueller, Phys. Rev. D **79**, 014017 (2009). https://doi.org/10.1103/PhysRevD.79.014017
52. A.V. Belitsky, D. Mueller, Phys. Rev. D **82**, 074010 (2010). https://doi.org/10.1103/PhysRevD.82.074010
53. A.V. Belitsky, D. Mller, Y. Ji, Nucl. Phys. B **878**, 214 (2014). https://doi.org/10.1016/j.nuclphysb.2013.11.014
54. X.D. Ji, J. Osborne, Phys. Rev. D **57**, 1337 (1998). https://doi.org/10.1103/PhysRevD.57.1337
55. A.V. Belitsky, D. Mueller, Phys. Lett. B **417**, 129 (1998). https://doi.org/10.1016/S0370-2693(97)01390-7
56. L. Mankiewicz, G. Piller, E. Stein, M. Vanttinen, T. Weigl, Phys. Lett. B **425**, 186 (1998). https://doi.org/10.1016/S0370-2693(98)00190-7
57. X.D. Ji, J. Osborne, Phys. Rev. D **58**, 094018 (1998). https://doi.org/10.1103/PhysRevD.58.094018
58. A.V. Belitsky, D. Mueller, L. Niedermeier, A. Schafer, Phys. Lett. B **474**, 163 (2000). https://doi.org/10.1016/S0370-2693(99)01283-6
59. A. Freund, M.F. McDermott, Phys. Rev. D **65**, 091901 (2002). https://doi.org/10.1103/PhysRevD.65.091901
60. A. Freund, M.F. McDermott, Phys. Rev. D **65**, 074008 (2002). https://doi.org/10.1103/PhysRevD.65.074008

61. A. Freund, M. McDermott, Eur. Phys. J. C **23**, 651 (2002). https://doi.org/10.1007/s100520200928
62. B. Pire, L. Szymanowski, J. Wagner, Phys. Rev. D **83**, 034009 (2011). https://doi.org/10.1103/PhysRevD.83.034009
63. H. Moutarde, B. Pire, F. Sabatie, L. Szymanowski, J. Wagner, Phys. Rev. D **87**(5), 054029 (2013). https://doi.org/10.1103/PhysRevD.87.054029
64. T. Altinoluk, B. Pire, L. Szymanowski, S. Wallon, (2012)
65. T. Altinoluk, B. Pire, L. Szymanowski, S. Wallon, JHEP **10**, 049 (2012). https://doi.org/10.1007/JHEP10(2012)049
66. M. Vanderhaeghen, P.A.M. Guichon, M. Guidal, Phys. Rev. D **60**, 094017 (1999). https://doi.org/10.1103/PhysRevD.60.094017
67. I.V. Anikin, B. Pire, O.V. Teryaev, Phys. Rev. D **62**, 071501 (2000). https://doi.org/10.1103/PhysRevD.62.071501
68. A.V. Radyushkin, C. Weiss, Phys. Rev. D **63**, 114012 (2001). https://doi.org/10.1103/PhysRevD.63.114012
69. N. Kivel, M.V. Polyakov, M. Vanderhaeghen, Phys. Rev. D **63**, 114014 (2001). https://doi.org/10.1103/PhysRevD.63.114014
70. A.V. Belitsky, D. Mueller, Nucl. Phys. B **589**, 611 (2000). https://doi.org/10.1016/S0550-3213(00)00542-3
71. V.M. Braun, A.N. Manashov, B. Pirnay, Phys. Rev. D **86**, 014003 (2012). https://doi.org/10.1103/PhysRevD.86.014003
72. V.M. Braun, A.N. Manashov, B. Pirnay, Phys. Rev. Lett. **109**, 242001 (2012). https://doi.org/10.1103/PhysRevLett.109.242001
73. A.V. Radyushkin, Phys. Rev. D **56**, 5524 (1997). https://doi.org/10.1103/PhysRevD.56.5524
74. J.C. Collins, A. Freund, Phys. Rev. D **59**, 074009 (1999). https://doi.org/10.1103/PhysRevD.59.074009
75. P. Kroll, H. Moutarde, F. Sabatie, Eur. Phys. J. C **73**(1), 2278 (2013). https://doi.org/10.1140/epjc/s10052-013-2278-0
76. I.I. Balitsky, A.V. Radyushkin, Phys. Lett. B **413**, 114 (1997). https://doi.org/10.1016/S0370-2693(97)01095-2
77. A.V. Radyushkin, Phys. Rev. D **59**, 014030 (1999). https://doi.org/10.1103/PhysRevD.59.014030
78. A.V. Belitsky, D. Mueller, Nucl. Phys. B **527**, 207 (1998). https://doi.org/10.1016/S0550-3213(98)00310-1
79. A.V. Belitsky, D. Mueller, Nucl. Phys. B **537**, 397 (1999). https://doi.org/10.1016/S0550-3213(98)00677-4
80. A.V. Belitsky, D. Mueller, A. Freund, Phys. Lett. B **461**, 270 (1999). https://doi.org/10.1016/S0370-2693(99)00837-0
81. A.V. Belitsky, D. Mueller, Phys. Lett. B **464**, 249 (1999). https://doi.org/10.1016/S0370-2693(99)01003-5
82. A.V. Belitsky, A. Freund, D. Mueller, Nucl. Phys. B **574**, 347 (2000). https://doi.org/10.1016/S0550-3213(00)00012-2
83. P. Hoodbhoy, X.D. Ji, Phys. Rev. D **58**, 054006 (1998). https://doi.org/10.1103/PhysRevD.58.054006
84. A.V. Belitsky, D. Mueller, Phys. Lett. B **486**, 369 (2000). https://doi.org/10.1016/S0370-2693(00)00773-5
85. A.V. Belitsky, A. Freund, D. Mueller, Phys. Lett. B **493**, 341 (2000). https://doi.org/10.1016/S0370-2693(00)01129-1
86. R. Brun, F. Rademakers, Nucl. Instrum. Methods Phys. **A389**, 81 (1997). https://doi.org/10.1016/S0168-9002(97)00048-X
87. URL https://root.cern.ch
88. M.R. Whalley, Comput. Phys. Commun. **57**, 536 (1989). https://doi.org/10.1016/0010-4655(89)90282-8
89. URL http://hepdata.cedar.ac.uk/pdf/pdf3.html
90. URL https://www.virtualbox.org
91. F. Hautmann, H. Jung, M. Krmer, P.J. Mulders, E.R. Nocera, T.C. Rogers, A. Signori, Eur. Phys. J. C **74**, 3220 (2014). https://doi.org/10.1140/epjc/s10052-014-3220-9
92. URL https://tmdlib.hepforge.org/doxy/html
93. URL http://www.ginac.de/CLN
94. URL http://www.qt.io
95. URL https://www.sfml-dev.org
96. URL http://www.oracle.com/technetwork/java/javaee/overview
97. S.V. Goloskokov, P. Kroll, Eur. Phys. J. C **42**, 281 (2005). https://doi.org/10.1140/epjc/s2005-02298-5
98. S.V. Goloskokov, P. Kroll, Eur. Phys. J. C **53**, 367 (2008). https://doi.org/10.1140/epjc/s10052-007-0466-5
99. S.V. Goloskokov, P. Kroll, Eur. Phys. J. C **65**, 137 (2010). https://doi.org/10.1140/epjc/s10052-009-1178-9
100. M. Vanderhaeghen, P.A.M. Guichon, M. Guidal, Phys. Rev. Lett. **80**, 5064 (1998). https://doi.org/10.1103/PhysRevLett.80.5064
101. M. Guidal, M.V. Polyakov, A.V. Radyushkin, M. Vanderhaeghen, Phys. Rev. D **72**, 054013 (2005). https://doi.org/10.1103/PhysRevD.72.054013
102. URL https://www.mysql.com
103. URL https://sqlite.org
104. A.V. Vinnikov, (2006)
105. C. Mezrag, H. Moutarde, F. Sabatié, Phys. Rev. D **88**, 014001 (2013). https://doi.org/10.1103/PhysRevD.88.014001
106. J.D. Noritzsch, Phys. Rev. D **69**, 094016 (2004). https://doi.org/10.1103/PhysRevD.69.094016
107. H. Moutarde, Phys. Rev. D **79**, 094021 (2009). https://doi.org/10.1103/PhysRevD.79.094021
108. K.A. Olive et al., Chin. Phys. C **38**, 090001 (2014). https://doi.org/10.1088/1674-1137/38/9/090001