Dynamic simulations in SixTrack

K. Sjobak, V.K. Berglyd Olsen, R. De Maria, M. Fitterer, A. Santamaría García, H. Garcia-Morales, A. Mereghetti, J.F. Wagner, S.J. Wretborn
CERN, Geneva, Switzerland

Abstract
The DYNK module allows element settings in SixTrack to be changed on a turn-by-turn basis. This document contains a technical description of the DYNK module in SixTrack. It is mainly intended for a developer or advanced user who wants to modify the DYNK module, for example by adding more functions that can be used to calculate new element settings, or to add support for new elements that can be used with DYNK.

Keywords
DYNK; SixTrack; Particle Tracking; Dynamic Kicks; fast failures; ripple.

1 Introduction

The goal of the DYNK module in SixTrack [1–4] is to make it possible to change element settings on a turn-by-turn basis. This feature was first implemented for simulating the action of scrapers in the SPS [5], and then re-implemented in a more general form as described in [6]. After this re-implementation, the module has been and is being used for simulations of crab cavity failures [7–10], asynchronous beam dumps [11], off-momentum collimation studies through RF detuning [12,13], studies of observed particle losses at the start of the energy ramp in the LHC [14, 15], and electron lens intensity modulation [16].

The use of the module is described in the SixTrack user manual [17]. For the most up-to-date version of the user manual, please see the source distribution of SixTrack [18]. This document describes the technical “inner workings” of the DYNK module, and is intended for those who want to extend the module with new mathematical functions or to handle new types of elements.

DYNK is designed to be extensible, and has already been extended several times by several people. It is the hope of the authors that this document will be helpful for those who wish to do this in the future. Note that it is strongly recommended to read about the usage of DYNK in the most recent version of the SixTrack user manual before reading this document.

2 Overview

The main code of DYNK is located in the Fortran module `dynk` found in the `dynk.s90` file. The `.s90` file format is free form Fortran (`.f90`) which will be pre-processed by ASTUCE[1]. All the data associated with DYNK is contained within the `dynk` module as saved arrays and variables.

Additionally, the subroutine `initialize_element` was added to the `daten` block. It is responsible for general initialization of `SINGLE ELEMENT`s at the end of the `daten` subroutine, whether DYNK is in use or not. It is also used to re-initialize such elements if they are later changed by DYNK.

The general program flow is illustrated in Figures 1 and 2. These show how DYNK is initialized, and how it is called. The structure is only shown for 6D thin tracking, however the pattern is the same for the other options.

[1]ASTUCE is a source code preprocessor that is used for SixTrack. It assembles the Fortran source files from one or more decks, typically containing a group of related subroutines; and blocks, small pieces of codes that are in-lined one or more places in one or more decks, typically used for defining Fortran COMMON blocks and kicks. Furthermore, the ASTUCE “pragmas” can include conditional statements, used to include or exclude parts of the source code based on compilation flags.

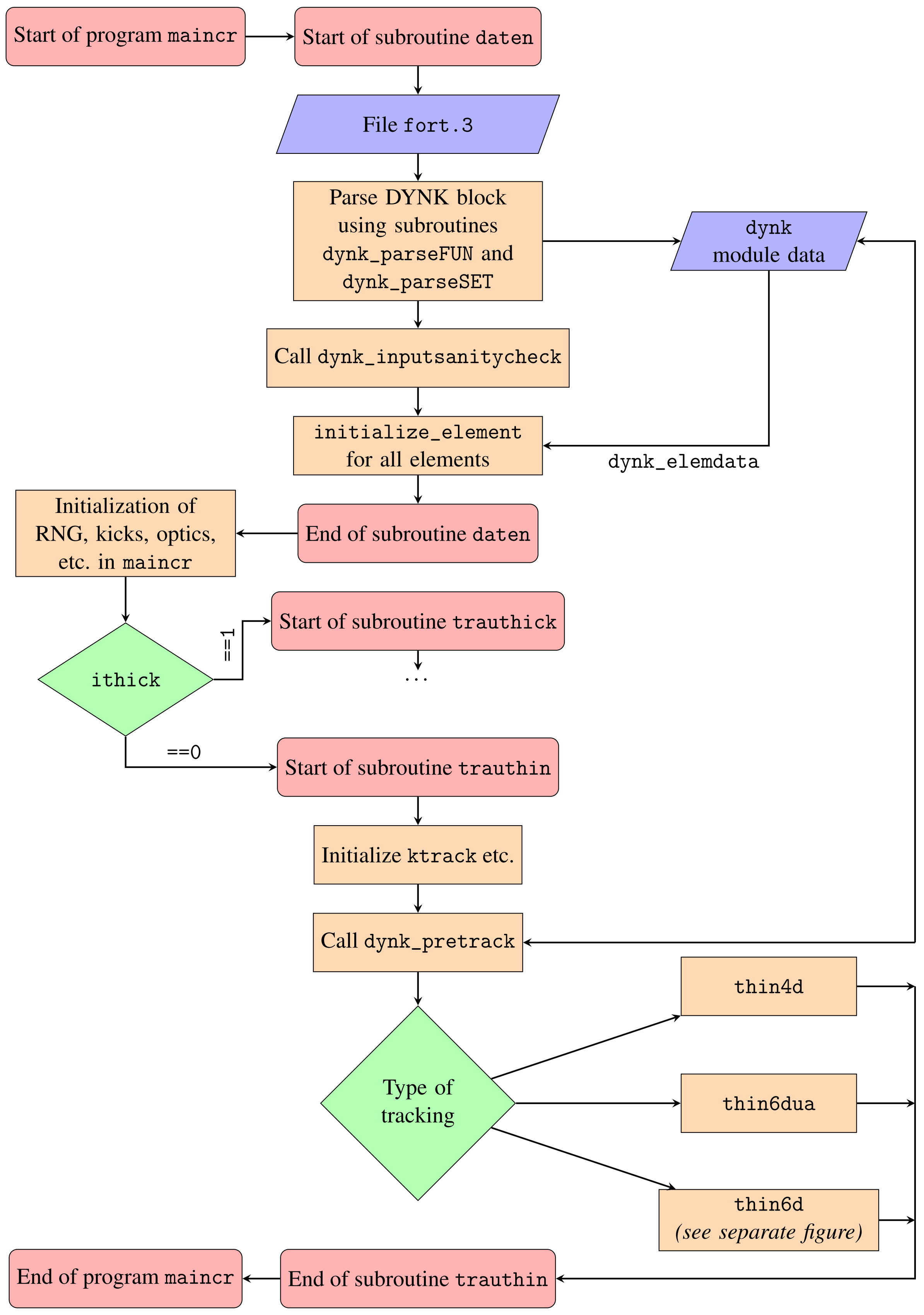

Start of program maincr
Start of subroutine daten
File fort.3
Parse DYNK block using subroutines dynk_parseFUN and dynk_parseSET
dynk module data
Call dynk_inputsanitycheck
initialize_element for all elements
dynk_elemdata
End of subroutine daten
Initialization of RNG, kicks, optics, etc. in maincr
ithick
==1
Start of subroutine trauthick
. . .
==0
Start of subroutine trauthin
Initialize ktrack etc.
Call dynk_pretrack
Type of tracking
thin4d
thin6dua
thin6d (see separate figure)
End of program maincr
End of subroutine trauthin

Fig. 1: Program flow in SixTrack with DYNK. The details of `thin6d` is shown in Figure 2. The beginning/end of processes are in red blocks, orange blocks are actions of interest, blue blocks are I/O, and green blocks are decisions.

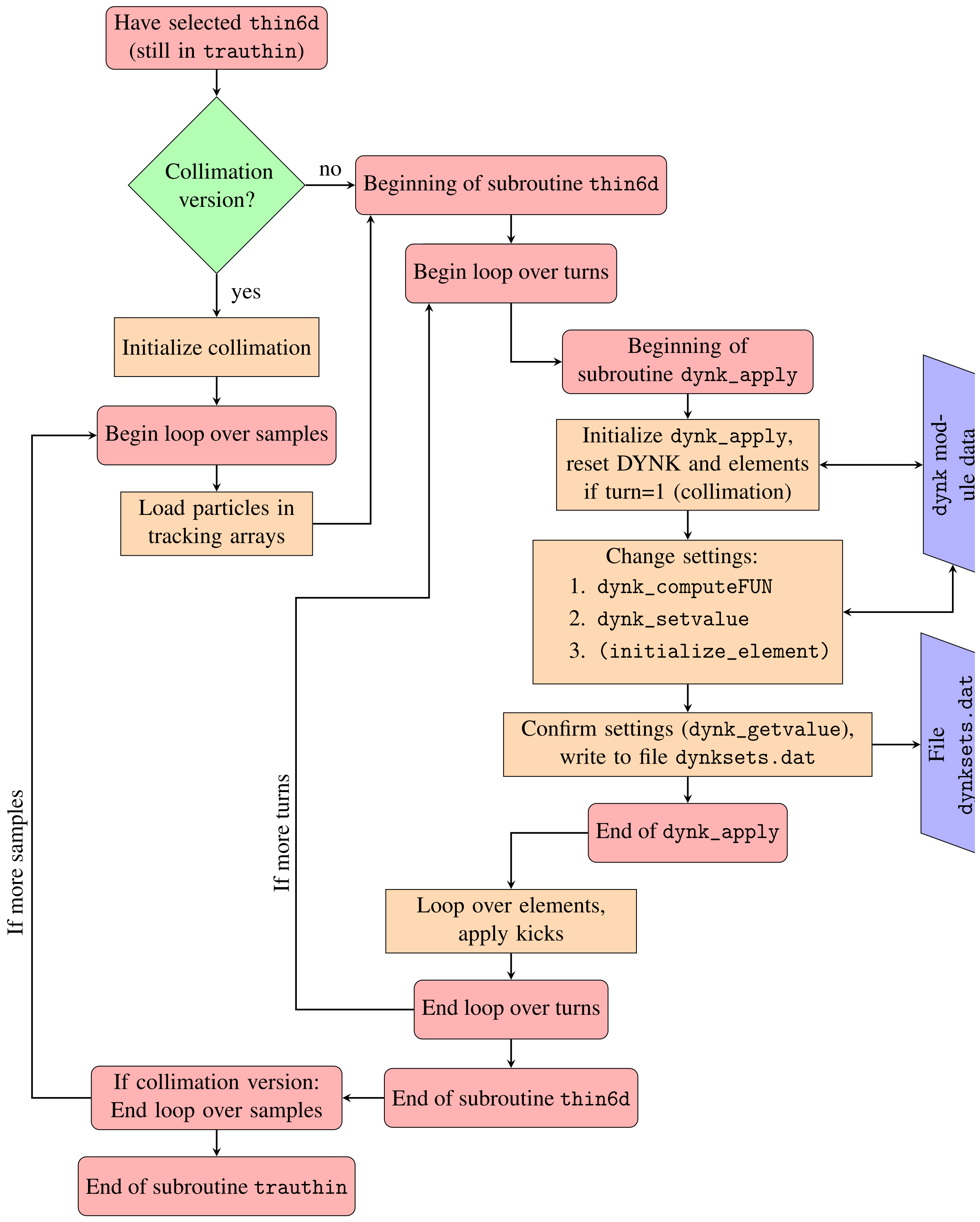

Have selected thin6d (still in trauthin)
Collimation version?
no
yes
Beginning of subroutine thin6d
Begin loop over turns
Initialize collimation
Beginning of subroutine dynk_apply
Begin loop over samples
Initialize dynk_apply, reset DYNK and elements if turn=1 (collimation)
dynk module data
Load particles in tracking arrays
Change settings:
1. dynk_computeFUN
2. dynk_setvalue
3. (initialize_element)
Confirm settings (dynk_getvalue), write to file dynksets.dat
File dynksets.dat
If more samples
If more turns
End of dynk_apply
Loop over elements, apply kicks
End loop over turns
If collimation version: End loop over samples
End of subroutine thin6d
End of subroutine trauthin

Fig. 2: Details of program flow when running subroutine `thin6d`; see Figure 1 for the rest of the program. The beginning/end of processes are in red blocks, orange blocks are actions of interest, blue blocks are I/O, and green blocks are decisions.

Note that all DYNK-related global variables have names postfixed with `_dynk`, while all functions and subroutines related to DYNK bear the prefix `dynk_`. Since the subroutine `initialize_element` is also used outside of DYNK, it does not contain this pre- or post-fix in its name. Furthermore, while this document is intended to give an overview of how DYNK functions, detailed descriptions of each subroutine, code stub, and variable are given in the code as comments.

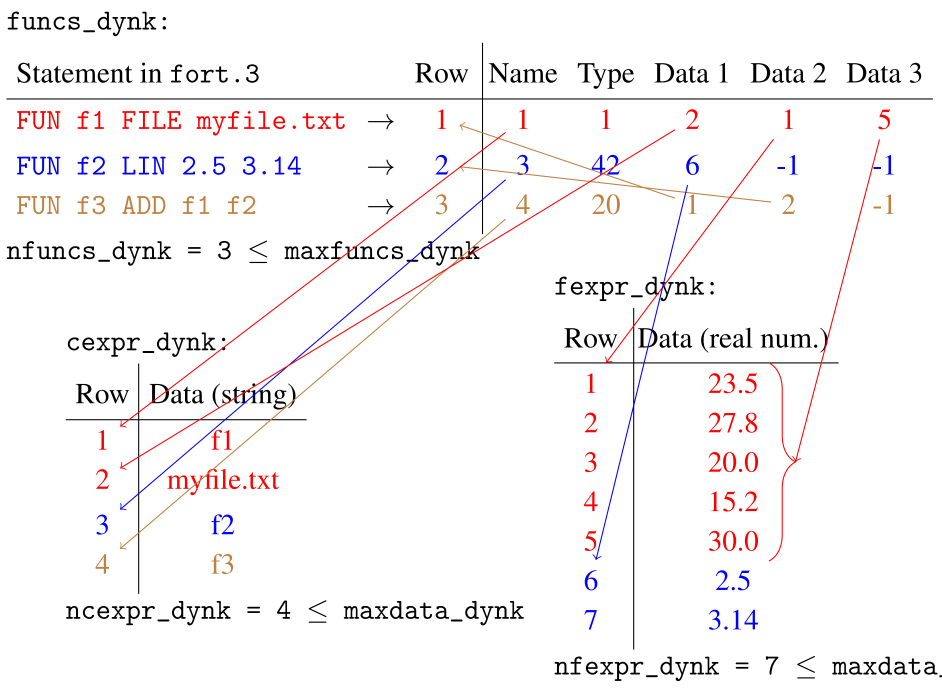
3 Data structure

The configuration of DYNK is stored in the `dynk` module. These variables and arrays are separated in 3 main categories, as described in the following subsections.

3.1 Overall configuration

Whether DYNK is active at all, i.e. if a DYNK block is present in the `fort.3` input file, is controlled by the `ldynk` boolean variable. If this is `.FALSE.`, no DYNK functions are called in the tracking loop. Furthermore, it is possible to enable extra debugging output and run-time checks (`ldynkdebug`), and to disable writing of the `dynksets.dat` output file (`ldynkfiledisable`).

3.2 Functions



`funcs_dynk`:

Statement in `fort.3`		Row	Name	Type	Data 1	Data 2	Data 3
`FUN f1 FILE myfile.txt`	→	1	1	1	2	1	5
`FUN f2 LIN 2.5 3.14`	→	2	3	42	6	-1	-1
`FUN f3 ADD f1 f2`	→	3	4	20	1	2	-1

`nfuncs_dynk = 3` $\leq$ `maxfuncs_dynk`

`cexpr_dynk`:

Row	Data (string)
1	f1
2	myfile.txt
3	f2
4	f3

`ncexpr_dynk = 4` $\leq$ `maxdata_dynk`

`fexpr_dynk`:

Row	Data (real num.)
1	23.5
2	27.8
3	20.0
4	15.2
5	30.0
6	2.5
7	3.14

`nfexpr_dynk = 7` $\leq$ `maxdata_dynk`

Fig. 3: Illustration of how three DYNK functions are stored in memory. The colours are used to separate the three functions used in the example (f1, f2, f3). The function f1 is of type FILE, where the column Data1 points to where in `fexpr_dynk` the data begins, and Data2 indicates how much data there is. The function f2 is of type LIN, where Data1 points to where in `fexpr_dynk` the two parameters are stored. The function f3 is of type ADD, where Data1 and Data2 indicates which functions should be added by listing their row numbers in `funcs_dynk`.

At the core of DYNK is the possibility to compute function values. This can for instance be used to change the strength of elements as a function of the turn number. The details of how this works is described in Section 5. As illustrated in Figure 3, the configuration for these functions is described in the table `funcs_dynk`, with one row per function (`FUN` statements in `fort.3`). Each of these rows consist of five integers, for which the first two have the same meaning for all function types. The meaning of the last three columns vary. Here, the first column always points to the location in the `cexpr_dynk` array where the (user-defined) name of the function is stored, while the second is an index which indicates the type of the function, and thus the interpretation of the following three columns. The number of function "slots" currently in use is defined by the variable `nfuncs_dynk` such that the last row that is currently in use is row number `nfuncs_dynk`[2]. The size of the table, and thus the maximum amount of slots, is given by the constant parameter `maxfuncs_dynk`.

Since many functions require the storage of more than three integers, DYNK has an internal memory allocation mechanism supporting the three main data types: integers, double precision real numbers, and strings. This is implemented as three large arrays `iexpr_dynk`, `fexpr_dynk`, and `cexpr_dynk`. Each string has a maximum length `maxstrlen_dynk`, and the stored strings are generally expected to be terminated and padded with binary zeros. Similar to `nfuncs_dynk`, each of these arrays have an associated integer that keeps track of how many elements are currently in use. These usage counters are called `niexpr_dynk`, `nfexpr_dynk`, and `ncexpr_dynk`, and should point to the last valid index in their corresponding data array.

It is the developer's responsibility to make sure that a new function (1) does not interfere with the data storage for other functions, and (2) does not exceed the available storage for this data type. For the first point, this means that the function must not modify the data stored by other functions, and it must correctly update the usage counters so that the next function to be defined can safely allocate more memory. In order to help with the second point, the subroutine `dynk_checkspace` is provided, which dynamically expands the arrays when needed. The expansion is normally done in chunks of 500 numbers or 200 strings. If more than this is requested, the allocation is exactly large enough to fit the requested amount.

3.3 Element settings

			`sets_dynk:`				`csets_dynk:`	
Statement in `fort.3`		Row	FUN idx.	First	Last	Shift	Element	Attribute
`SET magnet1 average_ms f3 1 5 0`	$\rightarrow$	1	3	1	5	0	magnet1	average_ms
`SET magnet1 average_ms f3 6 -1 -5`	$\rightarrow$	2	3	6	-1	-5	magnet1	average_ms
`SET crabcc1 voltage f2 3 10 -3`	$\rightarrow$	3	2	3	10	-3	crabcc1	voltage

`nsets_dynk = 3` $\leq$ `maxsets_dynk`

`sets_unique_dynk:`

Row	Element	Attribute
1	magnet1	average_ms
2	crabcc1	voltage

`nsets_unique_dynk = 2` $\leq$ `maxsets_dynk`

Fig. 4: Illustration of how three `SET` statements are stored in memory.

Similar to the functions, the element settings (`SET` statements in `fort.3`) are also stored in a few tables. However, since all settings are defined by the same fields, the memory management is considerably simpler. As illustrated in Figure 4 the main table is `sets_dynk`, which can store up to `maxsets_dynk` rows of integers. Each row of this table contains four columns. The first column is pointing to a row in the `funcs_dynk` array, indicating which function should be used for this element

[2]Note the Fortran convention of counting from 1.

setting. The two next columns indicate the range of turns for which the setting is active. The last column is an offset which is applied to the current turn number before computing the value of the function.

Additionally, there is a table `csets_dynk`, which has one row per `SET` with the same indexing as `sets_dynk`). Each row has two columns, both strings. The first column is the name of the `SINGLE ELEMENT` to be changed, and the second column is the name of the attribute that should be changed. This table is filled in the subroutine `dynk_pretrack`.

This architecture allows multiple functions to be defined for the same element, as long as the range of turns they are used for is not overlapping. This is verified by the `dynk_pretrack` subroutine, which also verifies that all `SET`s refer to valid elements and attributes. While this data structure makes it possible to apply different functions to the same element and attribute for different time periods, in many cases it is necessary to directly iterate over the element and attribute combinations that are used instead of the individual SET statements. To avoid excessive searching when iterating over unique elements and attributes, the table `csets_unique_dynk` is used. The table has the same format as `csets_dynk`, but has no duplicate entries. The number of rows used in this table is kept in the integer variable `nsets_unique_dynk`. Furthermore, the array `fsets_origvalue_dynk` contains the pre-changed value of the elements using the same indexing as `csets_unique_dynk`. This is necessary for collimation, where multiple samples of particles are tracked sequentially, making it necessary to be able to reset the state of the elements to how it was originally defined in `fort.2` and `fort.3`.

Finally, there are two arrays that are used to keep additional information on the level of structure elements, of which there may be several per single element. These arrays are named `dynk_izuIndex` and `dynk_elemdata`. The array `dynk_izuIndex` is used to store the index of the random number used to set the magnet error of the element in question. The array `dynk_elemdata` is used to store the various values needed to initialize certain types of elements, such as accelerating RF cavities, and will most likely be necessary for handling magnetic multipoles should this be implemented in the future.

4 Changing element settings: `dynk_apply`

If DYNK is active in the simulation, the subroutine `dynk_apply` is called in the tracking loop at the beginning of every turn, as shown in Figure 2. This subroutine then loops over the defined `SET`s, checking if it is active in the current turn. If this is the case, the current effective turn is calculated by adding the turn-shift (which is often 0) to the actual turn. The value of the active function is then calculated using `dynk_computeFUN` as described in Section 5.

Note that `dynk_apply` has an important role in initializing DYNK `FUN`ctions where the output depends on the internal state created by previous calls. This happens with the `FIR` and `IIR` filters, and also with the RNG-based functions `RANDG`, `RANDU`, and `RANDON`. This is done on the first turn, before any kicks are applied. The action taken is for example to initialize the "seed" that is updated at every call to the RNG function, with the initial seed read from `fort.3`.

4.1 The subroutine `dynk_setvalue`

When the new value has been calculated, it is applied to the specified element and attribute using `dynk_setvalue`. This function modifies the main element setting arrays in the same way as `daten` does when reading the input files, doing no calculations. In practice, this means that when changing single elements, it will write to the `ed`/`ek`/`el` and `elens_theta_max` arrays. The reason for this is to make the code, and its testing, as straight forward as possible.

One exception to the "no calculations" rule is that when changing the reference energy `E0` using `GLOBAL-VARS`, which triggers a recalculation of the reference momentum `e0f`, relativistic gamma `gammar`, and the energy-dependent particle arrays `dpsv`/`dpsv1`/`dpd`/`dpsq`/`oidpsv`/`rvv`.

4.2 The subroutine `initialize_element`

If any further calculation is needed, this is done afterwards in the subroutine initialize_element. This is for example used for nonlinear elements, where the actual kick strength is given by an average strength (`ed`) and a random component (`ek`). Furthermore, a unit conversion is also applied.

Another example is crab cavities, which store the phase offset in `el` when reading the input file, which is normally used to store the element length. This phase is therefore moved into a separate array `crabph`, which has the same shape and indexing as `el`.

Note that `initialize_element` is not only used when changing element settings, but also used just after reading the input files in `daten`. In this case, the second argument "`lfirst`" is set to `.TRUE.`.

4.3 The subroutine `dynk_getvalue`

The function `dynk_getvalue` returns the setting of an element's attribute as it was set by `dynk_setvalue`. This is used for output to `dynksets.dat`. When `ldynkdebug=.TRUE.` it is also used to confirm that the setting was correctly applied.

4.4 The virtual element `GLOBAL-VARS`

In order to change certain global settings, such as the reference energy `E0`, the element name `GLOBAL-VARS` is used. Because of this, it is not possible to have an actual element in the single elements list with this name when DYNK is active.

The `GLOBAL-VARS` element is also treated specially in the `dynk_setvalue` and `dynk_getvalue` subroutines. Here, this "element name" is tested for and handled near the top of the routine, before trying to locate the normal elements.

5 How the functions are calculated

As shown in Table 1, many types of functions are available in DYNK. The data used for their evaluation is stored in the `dynk` module, as described in Section 3.2. Internally, each function type is given an integer index, which is used in a `select case` when evaluating the function. Note that these indices are grouped by function type, and this grouping should be respected when adding new functions. A detailed description of the use of each function is given in the user manual, and how they store their data is documented in comments in the `dynk_parseFUN` code.

In most cases the DYNK `FUN`s are functions of the current turn number, including a possible turn-shift, which may be specified in the `SET` command. Note that DYNK functions can be chained such that one function can call another one, which may call yet another, etc. This is done by for example the `ADD` function, which calls two other functions, adds their results together, and returns the value. When this happens, the current turn number – including any turn-shift – is passed on to the next function.

5.1 Calculating the function values f(turn): `dynk_computeFUN`

The values of the functions are calculated in this Fortran function. It is called with an index into the relevant row in the `funcs_dynk` table, and the current (possibly shifted) turn number. Using the row index, it then reads the type index (second column), and executes the case containing the code for that function. The code may then read or modify any data in the `funcs_dynk` array and the `{c|f|i}expr_dynk` arrays. Note that it is expected that functions should only touch data in the arrays they themselves allocated in `dynk_parseFUN`, i.e. the functions should not interact outside of calling each other.

Index	Name	Short description
"System" functions:		
0	`GET`	Get original value of element/attribute.
1	`FILE`	Load from file (turn-by-turn).
2	`FILELIN`	Load from file (interpolate non-specified turns).
3	`PIPE`	Get setting from external program.
6	`RANDG`	Gaussian RNG.
7	`RANDU`	Uniform RNG.
8	`RANDON`	Random 1 or 0 with given probability.
Filters:		
10	`FIR`	Finite Impulse Response filter.
11	`IIR`	Infinite Impulse Response filter.
Operators (2-operand):		
20	`ADD`	Add the results of two other functions.
21	`SUB`	Subtract the results of two other functions.
22	`MUL`	Multiply the results of two other functions.
23	`DIV`	Divide the results of one function by the result of another.
24	`POW`	Exponentiate the result of one function with the result of another.
Operators (1-operand):		
30	`MINUS`	Result of another function, with opposite sign.
31	`SQRT`	Square root of the result from another function.
32	`SIN`	Sine of the result from another function.
33	`COS`	Cosine of the result from another function.
34	`LOG`	Natural logarithm of the result from another function.
35	`LOG10`	Base-10 logarithm of the result from another function.
36	`EXP`	Natural exponential function of the result from another function.
Polynomial functions:		
40	`CONST`	Constant value.
41	`TURN`	Current turn (after turn-shift).
42	`LIN`	Linear function of the turn number.
43	`LINSEG`	Linear function of the turn number (alternative input format).
44	`QUAD`	Quadratic function of the turn number.
45	`QUADSEG`	Quadratic function of the turn number (alternative input format).
Transcendental functions:		
60	`SINF`	Sine of the turn number, with specified ω, ϕ, and A.
61	`COSF`	Cosine of the turn number, with specified ω, ϕ, and A.
62	`COSF_RIPP`	Cosne of the turn number (alternate “RIPP” format).
Specialized functions:		
80	`PELP`	Parabolic-Exponential-Linear-Parabolic, used for energy ramping [20].
81	`ONOFF`	On for p_1 turns, then off for $p_2 - p_1$ turns, then repeat.

Table 1: Functions defined in DYNK and their indexes. For a full description, please see the user manual.

5.2 Initializing the functions: `dynk_parseFUN`

This function is responsible for parsing the user input (`FUN` statements), filling the `funcs_dynk` table, and allocating memory in the `{c|f|i}expr_dynk` arrays. It is one of the most complex parts of the DYNK module, and must be modified whenever one wants to add new DYNK functions. In `dynk_parseFUN`, each function type is initialized by a block of code. The selection of the code to execute depends on the function type listed in the second word of the relevant `FUN` statement. Note that all DYNK functions first call two support subroutines `dynk_checkargs` and `dynk_checkspace`, in order to check that the number of arguments is as expected for this function, and that there is enough space in the `{c|f|i}expr_dynk` arrays. If there is not enough space, it is automatically allocated, as described in Section 3.2.

Please also note that the initialization of the `GET` function, which returns the initial setting of an element, is only fully initialized after the `dynk_pretrack` subroutine has run.

6 Support for collimation version

The collimation version of SixTrack [19] is able to work around the 64 particle limit by tracking multiple samples of particles, i.e. loading the first 64 particles and then calling `thin6d` to track them for all turns, then loading the next 64 particles etc., as illustrated in Figure 2. For this to work correctly, DYNK must reset all element attributes at the beginning of each tracking simulation. This is accomplished in `dynk_apply`, which at the first turn of the first sample saves the original values (retrieved using `dynk_getvalue`), and then on the first turn of consecutive samples resets all elements and attributes touched by DYNK to the stored value. Furthermore, the outputfile `dynksets.dat` is only written while processing the first sample.

7 Checkpointing and restarting

The DYNK module supports checkpointing and restarting. For this to work, it must be able to truncate the `dynksets.dat` file to its position at the loaded restart point, and to store the current state of the DYNK functions. As all the current states of the functions are stored in the `{c|f|i}expr_dynk` arrays and in `fsets_dynk`, these arrays are written by the checkpoint/restart routines. Additionally, the array usage counters `n{c|f|i}expr_dynk` are also saved and restored. The rest of the state of DYNK is unchanged after initialization, and is thus recreated when reading the input files and initializing the simulations.

8 Summary and outlook

The DYNK module in SixTrack has been a success and has been in use for more than three years, enabling several new studies. Its overall architecture is designed to be extensible, to make it easy to add new functions, and to support acting on new types of elements and element attributes. The implementation of DYNK has also been taken as an opportunity to clean up several parts of the SixTrack code, as was done by implementing `initialize_element`. Furthermore, it not only supports the standard SixTrack, but also the collimation version and checkpoint/restart.

For the future, it is likely that several more functions and elements will be added, such as a wider variety of random distributions, and support for the beam-beam and multipole element. It may be interesting to support the setting of a global non-integer turn shift, in order to easily study the effects on different bunches along the ring. Furthermore, the performance of `dynk_setvalue` and `dynk_getvalue` might be improved by caching the element index, avoiding the search over all elements every time these functions are called. Finally, since DYNK can now change the reference energy using the `GLOBAL-VARS` mechanism, the large subroutines `thin6dua` and `thck6dua` can now in principle be removed, reducing the amount of code duplication.

References

[1] F. Schmidt, "SixTrack Version 4.2.16 Single Particle Tracking Code Treating Transverse Motion with Synchrotron Oscillations in a Symplectic Manner", CERN/SL/9456, 2012.

[2] G. Ripken and F. Schmidt, "A symplectic six-dimensional thin-lens formalism for tracking", DESY 95–63 and CERN/SL/95–12(AP), 1995.

[3] R. De Maria, A. Mereghetti, and K. Sjobak, "SixTrack Status", these proceedings.

[4] K. Sjobak, R. De Maria, E. McIntosh, A. Mereghetti, J. Barranco, M. Fitterer, V. Gupta, and J. Molson, "New features of the 2017 SixTrack release", IPAC'17, THPAB047.

[5] A. Mereghetti et al., "SixTrack-FLUKA active coupling for the upgrade of the SPS scrapers", IPAC'13, WEPEA064.

[6] K. Sjobak, H. Burkhardt, R. De Maria, A. Mereghetti, and A. Santamaría García, "General functionality for turn-dependent element properties in SixTrack", IPAC'15, MOPJE069.

[7] A. Santamaría García, H. Burkhardt, K. Hernández Chahín, A. Macpherson, K. Sjobak, D. Wollmann, and B. Yee-Rendón, "Limits on failure scenarios for crab cavities in the HL-LHC", IPAC'15, THPF095.

[8] K. Sjobak, R. Bruce, H. Burkhardt, A. MacPherson, A. Santamaría García, and Regina Kwee-Hinzmann, "Time Scale of Crab Cavity Failures Relevant for High Luminosity LHC" IPAC'16, THPOY043.

[9] A. Santamaría García, K. Sjobak, R. Bruce, H. Burkhardt, F. Cerutti, R. Kwee-Hinzmann, A. Lechner, and A. Tsinganis, "Machine protection from fast crab cavity failures in the High Luminosity LHC" IPAC'16, TUPMW025.

[10] R. Apsimon, G. Burt, A. Dexter, P. Baudrenghien, K. Sjobak, and R. Appleby, "Modelling the low level RF response on the beam during crab cavity quench", IPAC'17, MOPVA102.

[11] R. Bruce, C. Bracco, R. De Maria, M. Giovannozzi, A. Mereghetti, D. Mirarchi, S. Redaelli, E. Quaranta, B. Salvachua, "Reaching record-low $\beta*$ at the CERN Large Hadron Collider using a novel scheme of collimator settings and optics", Nuclear Instruments and Methods in Physics Research Section A, Volume 848, 11 March 2017, Pages 19–30, https://doi.org/10.1016/j.nima.2016.12.039.

[12] H. Garcia-Morales et al., "Simulating off-momentum loss maps using SixTrack", these proceedings.

[13] H. Garcia-Morales, R. Bruce, and B. Salvachua Ferrando, "Off-momentum loss maps with one beam", January 12th, 2016, CERN-ACC-NOTE-2016-0011, https://cds.cern.ch/record/2121307.

[14] S.J. Wretborn, R. Bruce, H. Garcia Morales, and K. Sjobak, "Study of off-momentum losses at the start of the ramp in the Large Hadron Collider", September 6th, 2017, CERN-ACC-NOTE-2017-0065, http://cds.cern.ch/record/2298696.

[15] J. Wretborn, R. Bruce, H.G. Morales, and K. Sjobak, "Off-momentum collimation at the start of the ramp" Presented at the 215th LHC Collimation Working Group Meeting, April 3rd 2017, https://indico.cern.ch/event/626908/.

[16] M. Fitterer, G. Stancari, A. Valishev, R. De Maria, S. Redaelli, K. Sjobak, and J.F. Wagner, "Implementation of hollow electron lenses in SixTrack and first simulation results for the HL-LHC" IPAC'17, THPAB041.

[17] F. Schmidt, A. Alekou, M. Fitterer, J.F. Wagner, S.J. Wretborn, R. De Maria, S. Kostoglou, K. Sjobak, and T. Persson, "SixTrack version 4.7.16: Single Particle Tracking Code Treating Transverse Motion with Synchrotron Oscillations in a Symplectic Manner; User's Reference Manual", please see
http://sixtrack.web.cern.ch/SixTrack/doc/manual_dev/six.pdf or

https://github.com/SixTrack/SixTrack/tree/master/Doc/user_manual
for the newest version.

[18] SixTrack sources, https://github.com/SixTrack/SixTrack/

[19] G. Robert-Demolaize, R. Assmann, S. Redaelli, and F. Schmidt, “A new version of SixTrack with collimation and aperture interface”, PAC’05, FPAT081.

[20] Stephan Russenschuck, “Field Computation for Accelerator Magnets: Analytical and Numerical Methods for Electromagnetic Design and Optimization”, Wiley-WCH 2010, ISBN 978-3-527-40769-9.