PLINK 2 File Format Specification Draft

February 27, 2019

The master version of this document can be found at `https://github.com/chrchang/plink-ng/tree/master/pgen_spec`.

Contents

1 Introduction

PLINK 1's binary genotype file format (described at `https://www.cog-genomics.org/plink/1.9/formats#bed`) is simple, compact, and supports direct computation on the packed data representation. Thanks to these properties, it continues to be widely used more than a decade after it was designed.

However, it can only represent unphased biallelic hard-called genotypes. This limitation makes it suboptimal for an increasing fraction of genome-wide association studies, which tend to benefit from inclusion of imputed genotype dosages and more sophisticated handling of multiallelic variants. In addition, the inability to represent phase information is problematic for investigation of compound heterozygosity, imputation-related data management, and several other workflows.

The most widely-used binary genotype file format which addresses these limitations is BCF (`https://github.com/samtools/hts-specs`). Unfortunately, while it is more machine-friendly than VCF, BCF does not support direct computation on packed data, so it's impossible to write software for it which can consistently match or exceed PLINK 1.9's efficiency.

Therefore, PLINK 2 introduces the PGEN binary genotype file format, a backward-compatible extension which can represent phased, multiallelic, and dosage data in a manner that provides much better support for "compressive genomics"[*]. It also incorporates SNPack[†]-style genotype compression, frequently reducing file sizes by 80+% with negligible encoding and decoding cost (and supporting some direct computation on the compressed representations). The result is, of course, not as simple as the PLINK 1 format, but the open-source `pgenlib_internal` library (used by PLINK 2) provides one way to ignore the additional complexity.

PLINK 2 also introduces backward-compatible extensions to PLINK 1's sample and variant text formats. PSAM (extension of .fam) supports storage of categorical (e.g. 1000 Genomes's "Population" field) and other phenotype / covariate data. PVAR (extension of .bim) supports storage of all header and variant-specific information in a typical VCF, and is defined such that "sites-only VCF" files are valid PVARs requiring no conversion by PLINK 2 at all.

2 PGEN Format Specification

PGEN is a binary format capable of representing mixed-phase, multiallelic, mixed-hardcall/dosage/missing genotype data.

A PLINK 1 variant-major .bed file is also a valid PGEN file. Yes, this is because it's grandfathered in as a special case. But it's a special case that doesn't take much additional code to handle, since the usual basic representation of hard-called biallelic genotype data is almost identical.

PGEN(+PVAR) is designed to interoperate with, not replace, VCF/BCF. PGEN cannot represent auxiliary per-genotype-call data such as read depths or quality scores; or biallelic genotype probability triplets; or triploid genotypes. It is just designed to represent the subset of the VCF format which is relevant to PLINK's domains of function. Considerable effort has been spent on optimizing PLINK 2's VCF ↔ PGEN conversion functions.

2.1 Overall file organization

A PGEN file is composed of a mandatory header, followed by a sequence of variant records. The header contains enough information to enable random access to the variant records.

A variant record's main data track can be "LD-compressed" (LD = linkage disequilibrium), referring to the most recent non-LD-compressed variant record and only storing genotype-category differences from it. This is the only type of inter-record dependency, so it doesn't take that much memory to process a PGEN file sequentially: other than the record type and size information in the header, only the genotypes from the latest non-LD-compressed variant might need to be kept for possible future reference.

A future version of this specification may add an optional footer, containing phase-set and possibly other information.

It is not always possible to write a PGEN file in a single sequential pass, since the header usually contains variant record lengths and compression methods which aren't known until variant-record-writing time. The `pgenlib_internal` implementation uses a two-pass approach, computing only the size of the header on the first write pass, and then backfilling it at the end. Three fixed-width storage modes are defined (covering basic unphased biallelic genotypes, unphased dosages, and phased dosages) which don't have this limitation, and are especially straightforward to read

[*] `https://www.nature.com/articles/nbt.2241`

[†] `http://sysbiobig.dei.unipd.it/?q=Software#SNPack`

and write; but they don't benefit from PGEN's low-overhead genotype compression. A future version of this specification may add a way to store most header information in a separate file, so that sequential reading, sequential writing, and genotype compression are simultaneously possible (at the cost of more annoying file management).

2.2 Header

2.2.1 Magic number

All PGEN files start with the two magic bytes `0x6c 0x1b`.

2.2.2 Storage mode

The third byte indicates the overall storage mode. The following modes are currently defined:

- `0x01` is PLINK 1 variant-major .bed. In this case, there are no more header bytes; the number of samples is not stored in the file, and must be known by some other means when initializing a PGEN reader.
- `0x02` is the simplest PLINK 2 fixed-width format. All variant records are of type `0`. (Variant record types are discussed in section 2.3.)
- `0x03` is the fixed-width unphased-dosage format. All variant records are of type `0x40`.
- `0x04` is the fixed-width phased-dosage format. All variant records are of type `0xc0`.
- `0x10` is the standard PLINK 2 format, with variable-width variant records.

Mode values `0x05..0x0f` and `0x11..0x7f` are reserved for use by future versions of this specification, and mode value 0 is off-limits since it corresponds to a PLINK 1 sample-major .bed file (and is detected and automatically converted to regular PGEN by PLINK 2). `0x80..0xff` can be safely used by developers for their own purposes.

2.2.3 Dataset dimensions, header body formatting

If the storage mode isn't `0x01`, the next eight bytes are the number of variants (as a little-endian `uint32`; everything else in this specification is also little-endian), followed by the number of samples as a `uint32` (call this N). This is followed by a byte describing how the rest of the header is formatted.

- Bits 0-3 of the twelfth byte indicate how variant-record types and lengths are stored. Interpreting them as a single number in 0..15, these are the meanings:
  - `0`: 4 bits per record type, 1 byte per record length.
  - `1`: 4 bits per record type, 2 bytes per record length.
  - `2`: 4 bits per record type, 3 bytes per record length.
  - `3`: 4 bits per record type, 4 bytes per record length.
  - `4`: 8 bits per record type, 1 byte per record length.
  - `5`: 8 bits per record type, 2 bytes per record length.
  - `6`: 8 bits per record type, 3 bytes per record length.
  - `7`: 8 bits per record type, 4 bytes per record length.
  - `8`: 0 bits per record type, 2 bits per record length ($\ell - \lceil N/4 \rceil$ is stored when the record length is ℓ byte(s)). Record type is inferred as 0 when record length is minimal, and 8 (i.e. multiallelic hard-call) when record length has 1-3 extra bytes. This is designed for single-sample files sharing a single .bim-like file.
  - `9`: 0 bits per record type, 4 bits per record length ($\ell - \lceil N/4 \rceil$ stored when record length is ℓ). Record type is inferred as 0 when record length is minimal, and 8 when record-length has 1-15 extra bytes.

  Values 10-15 are reserved for future use.

  With storage modes `0x02..0x04`, these bits are always zeroed out, since it is unnecessary to store any additional variant-record type or length information.
- Bits 4-5 indicate how many bytes are used to store each allele count. Zero indicates that no allele-count information is stored: either there are no multiallelic variants, or their allele counts must be known by some other means (e.g. loading the accompanying PVAR) before PGEN reading.

- Bits 6-7 indicate how "provisional reference" flag information is stored. `0` indicates that this information isn't in the PGEN (and should be present in the accompanying PVAR). `1` indicates that no reference alleles are provisional (all are trusted since they were e.g. imported from a VCF). `2` indicates that *all reference alleles are provisional* (none are trusted, usually because the PGEN was created from a PLINK 1 fileset which isn't designed to track REF/ALT alleles). `3` indicates that some but not all reference alleles are provisional, and this is tracked by bitarrays in the header.

Storage modes defined in the future may use more than one byte to describe header formatting.

2.2.4 Variant block offsets

For the variable-width storage mode (`0x10`), the PGEN file is divided into "variant blocks" of 2^{16} variants. (The last block may contain fewer variants.) Thus, if there are M variants, there are $B := \lceil M/2^{16} \rceil$ variant blocks. The next $8B$ bytes of the header store the (0-based) offsets of each variant block within the file, as `uint64`s.

This subsection is not present for the fixed-width modes.

2.2.5 Main header body

For the variable-width storage mode (`0x10`), the header body is also divided into blocks corresponding to 2^{16} variants each. Each block contains a packed array of 2^{16} variant record types, followed by a packed array of variant record lengths, possibly followed by an array of allele counts, and then possibly followed by a provisional-REF-allele flag bitarray. All of these arrays end at byte boundaries; when there are unused bits at the end of any of these arrays, they must be set to zero. This is also true for all packed arrays of 1/2/4-bit elements mentioned later in this specification.

For the fixed width storage modes, only the provisional-REF-allele flag bitarray might be present. (It can be interpreted as being divided into blocks corresponding to 2^{16}, or not; doesn't make a difference.)

2.2.6 Example

Suppose we're generating a standard PGEN from a PLINK 1 fileset with 1092 samples and 39728178 variants.

- Bytes 0-2 are `0x6c 0x1b 0x10`, the magic number followed by a byte indicating the usual compression-supporting storage mode. (In the rest of this specification, phrases like "byte *a*-*b*", "byte *c*", "record #*d*", and "block #*f*" use 0-based indexing. Only "first"/"second"/"third"/"*n*th" uses 1-based indexing.)

- Bytes 3-6 are `0x32 0x34 0x5e 0x02`, the little-endian binary representation of 39728178.

- Bytes 7-10 are `0x44 0x04 0x00 0x00`, the little-endian binary representation of 1092.

- Byte 11 is probably `0x81`, indicating 4 bits per record type, 2 bytes per record length, no allele counts (they're unnecessary since all variants are biallelic), and no trusted REF alleles (since we're generating this from a PLINK 1 fileset where REF/ALT may not be tracked). It would be `0x41` if the PLINK 1 fileset always had A2=REF and we made PLINK 2 aware of this with the `--real-ref-alleles` flag.

- Bytes 12-19 store the starting position of variant record #0 in the file; bytes 20-27 store the starting position of variant record #65536; . . . ; bytes 4860-4867 store the starting position of variant record #39714816.

- Bytes 4868-37635 store the first 2^{16} variant record types. The low 4 bits of byte 4868 are the type of record #0, the high 4 bits of byte 4868 are the type of record #1, the low 4 bits of byte 4869 are the type of record #2, etc.

- Bytes 37636-168707 store the first 2^{16} variant record byte lengths.

- Bytes 168708-201475 store the types of variant records 65536-131071, and bytes 201476-332547 store the lengths; . . . ; bytes 99291908-99298588 store the types of variant records 39714816-39728177, and bytes 99298589-99325312 store the lengths.

- Variant record #0 starts at byte 99325313. Thus, bytes 12-19 are `0x81 0x95 0xeb 0x05 0x00 0x00 0x00 0x00`.

Note that the variant-block starting positions stored in bytes 12-4867 make it possible to extract an arbitrary variant record from the middle without reading most of the 99.3 MB header. For example, to extract record #31000000, it's useful to read bytes 3796-3803 to get the starting position of record #30998528; it's necessary to read byte 77501924 to get the variant record type; and it's necessary to read bytes 77533956-77536902 to get the lengths of *every* record from #30998528 to #31000000, since these lengths are needed to compute the start and end positions of record #31000000 given the start position of record #30998528. This last part is annoying, but it would be far more annoying if we had to add 31 million numbers together instead of less than 1500 of them.

2.3 Variant records

Each variant record starts with the main data track, which is sufficient to describe unphased biallelic hard-calls. The following 10 auxiliary data tracks may also appear (in the listed order), depending on the variant record type:

1. Multiallelic hard-calls.
2. Hardcall-phase information.
3. Biallelic dosage existence.
4. Biallelic dosage values.
5. Multiallelic dosage existence.
6. Multiallelic dosage values.
7. Biallelic phased-dosage existence.
8. Biallelic phased-dosage values.
9. Multiallelic phased-dosage existence.
10. Multiallelic phased-dosage values.

If the file is PLINK 1 variant-major .bed (storage mode `0x01`), all variant records are of PLINK 1's type, which doesn't have an associated numeric code, and no auxiliary data tracks may be present.

Otherwise, bits 0-2 of the variant record type describe how the main data track is stored, bit 3 indicates whether multiallelic hard-calls are present, bit 4 indicates whether hardcall-phase information is present, bits 5 and 6 describe whether and how unphased dosage data is stored, and bit 7 indicates whether explicit phased-dosages are present.

Variant records are currently limited to 4284736160 (slightly under 2^{32}) bytes.

2.3.1 Difflists

Before describing the main data track in more detail, it is useful to define what a PGEN "difflist" is. This construct appears several times in the subsections to follow.

A difflist represents either a possibly empty increasing sequence of sample IDs, or a possibly empty sequence of (sample ID, 2-bit genotype category value) pairs. (Sample ID values in PGEN files are always integers in $[0, N-1]$, where N is the number of samples.) As the name suggests, it is designed to represent a sparse list of differences from something else. It does so in a manner that is compact, and supports fast checking of whether a specific sample ID is in the list.

All difflists start with a base-128 varint (see `https://developers.google.com/protocol-buffers/docs/encoding#varints`), indicating how many elements are in the list; call this value L. If $L = 0$, that's the end of the list.

Otherwise, the elements are divided into groups of 64 (the last group may be smaller); call the number of groups $G := \lceil L/64 \rceil$. The remaining components of the difflist are, in order:

1. Sample IDs #0, #64, #128, #192, . . . (i.e. the first ID in each group), stored as `uint8`s iff $N \leq 2^8$, `uint16`s iff $2^8 < N \leq 2^{16}$, `uint24`s iff $2^{16} < N \leq 2^{24}$, and `uint32`s otherwise.
2. $G - 1$ bytes, where byte j in this component is 63 less than the byte size of group #j in the final component. (63 is subtracted because the minimum possible raw value is 63, and the maximum possible value is larger than 255 but not larger than 255+63.) To reduce overhead for short lists, the byte size of the last group is not stored in this manner.

3. If 2-bit genotype category values are also stored in the difflist, the fourth component is a packed array of them, occupying $\lfloor L/4 \rfloor$ bytes.

4. The final component is a sequence of $L - G$ varints encoding ([sample ID #k] - [sample ID #$k-1$]) for all positive $k < L$ which aren't multiples of 64.

As an example, suppose $N = 488377$, $L = 79$, 2-bit genotype category values are also stored in the difflist, and the sample IDs in the list are 5000, 10000, 15000, ..., 395000. Then:

- The first byte of the difflist is `0x4f`, 79 encoded as a varint (which is just plain 79, since $79 < 128$).
- The next six bytes store sample IDs #0 and #64 in the list, using `uint24` encoding (i.e. `uint32` with the last byte omitted). `0x88 0x13 0x00 0x88 0xf5 0x04`.
- The next byte is the final-component byte size of the first group (we'll see that this is 126) minus 63. `0x3f`.
- The next 20 bytes store the packed genotype category values associated with the sample IDs in the difflist. More precisely, the bottom two bits of the first byte are for sample #5000, bits 2-3 of the first byte are for sample #10000, bits 4-5 of the first byte are for sample #15000, bits 6-7 of the first byte are for sample #20000, bits 0-1 of the second byte are for sample #25000, ...
- The remainder of the difflist stores ([sample ID #1] - [sample ID #0]), ([sample ID #2] - [sample ID #1]), ..., ([sample ID #63] - [sample ID #62]), ([sample ID #65] - [sample ID #64]), ([sample ID #66] - [sample ID #65]), ..., ([sample ID #78] - [sample ID #77]) using varint encoding. There are 77 values (note that there is no entry for $k = 64$), all equal to 5000 in this example, so these 154 bytes are `0x88 0x27 0x88 0x27 0x88 0x27 ...`, since 5000's varint encoding is `0x88 0x27`.

2.3.2 Main data track

The main data track distinguishes between homozygous-REF, heterozygous REF-ALT, double-ALT, and missing hard-calls. These 4 categories are sufficient to describe all unphased biallelic hard-call possibilities, while still making sense for multiallelic variants.

If the variant record is of PLINK 1's type, the four categories are encoded in a packed array of 2-bit values as `0` = double-ALT, `1` = missing, `2` = heterozygous REF-ALT, and `3` = homozygous-REF. Otherwise, the 2-bit encoding is `0` = homozygous-REF, `1` = heterozygous REF-ALT, `2` = double-ALT, and `3` = missing, and the bottom three bits of the variant record type indicates what compression is used on the resulting packed array.

- `0`: no compression.
- `1`: "1-bit" representation. This starts with a byte indicating what the two most common categories are (value `1`: categories 0 and 1; `2`: 0 and 2; `3`: 0 and 3; `5`: 1 and 2; `6`: 1 and 3; `9`: 2 and 3); followed by a bitarray describing which samples are in the higher-numbered category; followed by a difflist with all (sample ID, genotype category value) pairs for the two less common categories.
- `2`: LD-compressed. A difflist with all (sample ID, genotype category value) pairs for samples in different categories than they were in in the previous non-LD-compressed variant. The first variant of a variant block (i.e. its index is congruent to 0 mod 2^{16}) cannot be LD-compressed.
- `3`: LD-compressed, inverted. A difflist with all (sample ID, genotype value) pairs for samples in different categories than they would be in the previous non-LD-compressed variant after inversion (categories 0 and 2 swapped). This addresses spots where the reference genome is "wrong" for the population of interest.
- `4`: Difflist with all (sample ID, genotype category value) pairs for samples outside category 0.
- `5`: Reserved for future use. (When all samples appear to be in category 1, that usually implies a systematic variant calling problem.)
- `6`: Difflist with all (sample ID, genotype category value) pairs for samples outside category 2.
- `7`: Difflist with all (sample ID, genotype category value) pairs for samples outside category 3.

2.3.3 Multiallelic hard-calls

When bit 3 of the variant record type is set, multiallelic hard-call(s) are present involving an ALT allele after the first. In this case, all category 1 hard-calls are initially assumed to be REF-ALT1 (ALT1 = first ALT allele), and category 2 hard-calls are initially assumed to be homozygous-ALT1; then the "patch sets" described below are applied to fill in hard-calls involving ALT2/ALT3/etc.

The low 4 bits of the first byte of auxiliary data track #1 indicates how the category 1 patch set is formatted.

- `0`: Patch set starts with a bitarray with J bits, where J is the number of samples in genotype category 1, and bit $j \in [0, J)$ is set iff the $(j+1)$th smallest sample ID among those in genotype category 1 isn't a REF-ALT1 genotype. Let K be the number of set bits; then the bitarray is followed by a packed array of K fixed-width values, where the width depends on the total number of ALT alleles, and each value is 2 less than the ALT allele index in the corresponding genotype:
  - 2 ALT alleles: width ZERO. All set bits in the initial bitarray correspond to REF-ALT2.
  - 3 ALT alleles: width 1 bit. I.e. set bits in the packed array correspond to REF-ALT3, clear bits correspond to REF-ALT2.
  - 4-5 ALT alleles: width 2 bits.
  - 6-17 ALT alleles: width 4 bits.
  - 18-257 ALT alleles: width 8 bits.
  - 258-65537 ALT alleles: width 16 bits.
  - 65538-16777215 ALT alleles: width 24 bits.
- `1`: Same as format 0, except that the initial bitarray is replaced with a difflist containing the sample IDs with category 1, not-REF-ALT1 genotypes.
- `2..14`: These values are reserved for future use.
- `15`: Empty patch set; everything in category 1 is REF-ALT1.

The high 4 bits of the first byte indicates how the category 2 patch set is formatted. The actual patch set appears after the end of the category 1 patch set.

- `0`: Patch set starts with a bitarray with J bits, where J is the number of samples in genotype category 2, and bit $j \in [0, J)$ is set iff the $(j+1)$th smallest sample ID among those in genotype category 2 isn't a homozygous-ALT1 genotype. Let K be the number of set bits. Then, if there are exactly 2 ALT alleles, this is followed by a bitarray with K bits where set bits correspond to homozygous-ALT2 genotypes, and clear bits correspond to heterozygous ALT1-ALT2. Otherwise, the initial bitarray is followed by a packed array of K pairs of fixed-width values, where the width depends on the total number of ALT alleles, each value is 1 less than the ALT allele index in the corresponding genotype, and the first value in each pair is never larger than the second:
  - 3-4 ALT alleles: width 2 bits (so each genotype requires 4 bits). For example, an ALT1-ALT3 heterozygous call would be represented by the 4-bit value 8: the high 2 bits are equal to 2, representing ALT3, and the low 2 bits are equal to 0, representing ALT1.
  - 5-16 ALT alleles: width 4 bits (each genotype is 8 bits).
  - 17-256 ALT alleles: width 8 bits (each genotype is 16 bits).
  - 257-65536 ALT alleles: width 16 bits (each genotype is 32 bits).
  - 65537-16777215 ALT alleles: width 24 bits (each genotype is 48 bits).
- `1`: Same as format 0, except that the initial bitarray is replaced with a difflist containing the sample IDs with category 2, not-homozygous-ALT1 genotypes.
- `2..14`: These values are reserved for future use.
- `15`: Empty patch set; everything in category 2 is homozygous-ALT1.

2.3.4 Phased heterozygous hard-calls

When bit 4 of the variant record type is set, phased heterozygous hard-calls are present.

In this case, the first *bit* of auxiliary data track #2 indicates whether an explicit "phasepresent" bitarray is stored. Let H be the number of heterozygous hard-calls; when this initial bit is set, the next H bits (including the high 7 bits of the first byte) are 1-bit values where 0 = no hardcall-phase for that heterozygous call, and 1 = the call is phased. When the initial bit is clear, all heterozygous hard-calls are phased.

This is followed by a "phaseinfo" bitarray, which begins at the usual byte boundary when "phasepresent" is stored, but begins at bit 1 of the first auxiliary data track #2 byte instead when phasepresent is omitted. It has P bits, where P is the number of phased heterozygous hard-calls; clear bits correspond to "unswapped" alleles (first allele index is less than the second, like "0|1" in the VCF GT field), and set bits correspond to "swapped" alleles.

Note that the PGEN format does not distinguish between GT values of "0/0" and "0|0", since the distinction is meaningless. (Phase sets are meaningful, but they can't consistently be represented by switching between "0/0" and "0|0" anyway.)

2.3.5 Biallelic dosage existence and values

When bit 5 of the variant record type is set, and bit 6 is clear, auxiliary data track #3 is a difflist indicating which samples have dosage information in auxiliary data track #4, and auxiliary data track #4 contains 16-bit ($0..2^{15}$; 65535 missing value is only permitted in the unconditional-dosage case mentioned next) values expressing the sums of all ALT allele dosages for those samples.

When bit 5 of the variant record type is clear, and bit 6 is set, dosage is stored for all samples. Auxiliary data track #4 is as described above, and there is no track #3.

When bits 5 and 6 of the variant record type are both set, auxiliary data track #3 is a bitarray with N bits, where a set bit indicates a dosage in track #4. Track #4 is as described above.

For example, suppose $N = 488377$, every even sample ID has an ALT allele dosage of 0.75 on a 0..2 scale, and no odd sample IDs have associated dosages. Then a bitarray is a more compact way to represent the sample ID list than a difflist, so bits 5 and 6 of the variant record type would normally be set. Auxiliary data track #3 would contain 61047 bytes with value `0x55`, followed by a byte with value `0x01`. Auxiliary data track #4 would be `0x00 0x30 0x00 0x30 0x00 0x30 ...`, containing 244189 16-bit values all equal to $(0.75/2) \cdot 2^{15}$.

2.3.6 Multiallelic dosages

TBD.

2.3.7 Phased dosages

TBD.

2.3.8 Phased multiallelic dosages

TBD.

3 PSAM Format Specification

TBD.

4 PVAR Format Specification

TBD.