Insight2 Software Architecture

It is expected that Insight parameter refinement will be run in an inner loop of an optimization algorithm designed to identify informative covariates, such as FitCons, and as a result, performance is important.

The original algorithm employed a series of scripts to transform a genomic database and a BED file indicating the genomic positions of interest, into a INS file that could be processed by the INSIGHT code. For a typical 10Mbase BED file, it would take 20 minutes for these scripts to generate the INS file, and another 5 minutes for INSIGHT to generate maximum likelihood values for the parameters. Thus 80% of runtime was dedicated to data preparation and 20% to numerical inference. Theoretical runtime should have been at least 1,000x faster and was dominated by the function needed to accumulate large number of probabilities, without underflowing the efficient machine precision for floating point numbers.

The original C++ version of the INSIGHT code (INSIGHT1, Ilan) implemented an expectation maximization algorithm that was likely considered necessary due to the presence of latent variables, in particular the recent ancestral allele at each genomic position (). However, the imposition of the infinite sites model allowed a characterization of input data based on properties of the relationship between the deep ancestral allele () and the observed distribution of human alleles (). This characterization removed the explicit dependency on and allowed calculation of input data likelihood based directly from the model parameters. Once this was available, EM became unnecessary and EM inference of parameters was replaced with a bounded gradient descent search on likelihood surface.

Reorganization of this code proceeded in five phases

1. Limiting data set to human reference values
2. Converting ASCII databases into a binary storage format
3. Improving efficiency and flexibility by replacing EM with direct Maximum likelihood inference using a bounded gradient-based method (lgbfs). This also allowed the inclusion of weighted observations to accommodate mixture models.
4. Extending the code to incorporate priors and generate
   1. maximum aposteriori probability estimates for parameters
   2. full posterior parameter distributions
5. Calculation of posterior probabili9ty estimates or each genomic position, including
   1. Posterior estimates of selection classes and variables (S,A,W)
   2. Posterior allele probabilities at each position
   3. Counterfactual analysis of selection variables for each allele (experimental)

Phase 1: Limiting the data

A complete input to the INSIGHT model requires consideration of :

1. Set of neutral polymorphic positions.
2. A non-overlapping near partition of the genome into Blocks (~5Kbp, contiguous ) of similarly evolving genomic positions.
3. An estimate of mean divergence rate () for each block.
4. An expected distribution of deep ancestral alleles () over nucleotides A,C,G,T at each genomic position.
5. A prior distribution of proximal ancestral alleles () over nucleotides A,C,G,T at each genomic position.
6. Observed human allele frequencies () over A,C,G,T at each genomic position.
7. Model parameters and block parameter can be estimated from these

This would require approximately 12 values per genomic position (4 for each of , , and ) making the database on the order of 3billion positions \* 12 values/position = 36 billion values. At a naive 8 bytes per double precision value this would be 288GBytes. Even with a more efficient encoding of 2 bytes per value this is 72GB, too large to conveniently fit in memory for commonly available servers in 2015, and slow to load from external storage.

To accelerate this computation, the database size is reduced to < 5GB of memory so it can be loaded into main memory and accessed rapidly when accessed in order of increasing genomic positions. The following steps provide this preprocessing.

The block structure (2) and set of neutral polymorphic sites (1) are for the human data set based on the human reference HG19 and variation in the Complete Genomics set of 54 unrelated individuals.

Polyfen (?) is then used to estimate for each block (3) and the distribution over (4) for each genomic position, based on alignments of the reference genomes of chimpanzee (PanTro), orangutan (PonAbe) and Rhesus Macaque (RheMaq) to the human reference.

The infinite sites assumption is applied to decedents of from . Therefore any site with more than 2 different alleles is in is excluded from analysis. This reduces the relationship between , , and to two properties that are sufficient for data likelihood calculation under INSIGHT (Table3):

SelectionClass (approximation to the site frequency spectrum of ):

* Monomorphic (Mono or M),
* Polymorphic at Low Frequency (MAF<=15%) (PolyL or L)
* Polymorphic at High Frequency (MAF>15%) (PolyH or H)

AncestralAllele:

* [ZeqXmaj],
* [ZeqXmin],
* [ZeqXoth].

These sufficient properties reduce the number of values per position to a maximum of 2, a selection class, and two probability values. As each block requires only two additional values ( and ) to represent each 5000 positions, the additional memory overhead is three orders of magnitude smaller, so is neglected.

As polymorphic sites are relatively rare (1/1000 the number of monomorphic sites), and monomorphic sites have no minor allele, this reduces the typical data storage requirement for a site to that of a single monomorphic site: 2 bits for the selection class, and a single double precision value for , as and

Phase 2: Conversion to Binary

The process of converting ASCII floating point values to binary provides two benefits

* Dramatic reduction in data loading and interpretation time
* Reduction in data redundancy via indirection

Even at one single precision floating point value per genomic position, (3Gbase \* 4 bytes / base), 12GB of memory would be requires to store the data for all genomic positions. While careful truncation of probability resolution might reduce this to 2 bytes per base, even this proved unnecessary. As the human allele distribution was derived from 54 individuals, only 240 unique values for and were produced, across the entire genome. This allowed the probability value to be replaced with a single byte index into a 256 entry lookup table of double precision values, reducing the memory footprint to 1 byte / genomic position or around 3GB without sacrificing numerical precision. The index value of 0 was used to indicate missing data.

Conversion to a binary database progressed in three steps

* Identification of unique values of and generation of lookup table
* Separation of genomic positions Mono and Poly subsets
* Encoding of each subset into a binary file format, with a dense Mono database, having one byte for each position in autosomal HG19, and a sparse Poly database, with a chromosome, position Allele frequency and 2 probability indices for each position.

Monomorphism data is stored in a binary-specified format, while polymorphism data is read from ASCII formatted files and automatically translated to a “.cache” file that represents the same data in hosts memory organization. If present, the “.cache” file is read directly into memory via a single IO call, rather than reinterpreted from ASCII.

Code Support:

* Butils:fastfile.h High speed asci -> float / int / string interpretation, more than 10x  
   faster than stdio.
* butils:RTable.h Storage and conversion between table indexes and float / string data
* butils:vecarray.h Conversion from, ASCII-friendly dynamic vectors, to fixed size arrays  
   suitable for binary IO, “.cache” file generation, and source selection.

Phase 2a: Database access and Model Primitives

The C++ namespace **insdb:** provides top-level access to the insight database and can be simply modified encode indices using 2 bytes if more than 255 unique probability values are needed. Conceptually, the database is composed of 4 components:

* Mono – an array of indices into the probability database representing P(ZeqXmaj), 1 for each genomic position with a 0 index indicating missing data, or a non-monomorphic position.
* Block – an ordered array of non-overlapping INSIGHT block boundaries providing: chromosome, start, end, lambda, and theta values for the corresponding block.
* Poly – an ordered array of polymorphic sites, with each element containing: chromosome, position, Frequency Class (H/L), and 2 indices into the probability table providing P(ZeqXmaj) and P(ZeqXmin).
* PolyN – A subset of the Poly database representing only neutral polymorphic sites.

The database is read by providing string indicating the database location in a file system to **DB:ReadDB()**. The database is usually traversed in ascending order (as per *.bed* file sort order) by initializing the traversal using DB:locStreamInit(), followed by a succession of **locStreamFast()** or **locStreamBig()** for detailed, or summary data, respectively.

Efficient representation of database subsets is provided by the **inscomp::Block** template. Rather than maintain a list of sites, each with its own sufficient properties, Block maintains a list counts for each set of sufficient properties. This can be far more compact and efficiently enables dynamic weighting of genomic positions as a fractional number of observations of corresponding sufficient properties.

The C++ namespace **inscomp:** holds structures to load and manipulate INSIGHT models and parameters, including simple calculation of data likelihood (for debugging), and converting data base subsets into a high-performance format for the rapid calculation of likelihoods / MaP values that might be used in generating optimal values, or posterior parameter distributions. **Inscomp::modelInsight** is the central class for this namespace and base class for many of its members. **modelInsight::likelihood()** and **modelInsight::likelihoodMatrix()** calculate both likelihood of an observation (based on its sufficient properties), and expected probability distribution over each pattern of sufficient properties.

Phase 3: EM to ML

Imposition of the infinite sites model on the descent of from allowed the explicit calculation of data likelihood from sufficient properties, and eliminated the need to use Expectation Maximization to explicitly model distributions over latent variable . The result is that the likelihood of one observation under the model can be reduced to a simple nonlinear function in global model parameters ,, in the form

This relatively smooth function is amenable to direct numerical optimization via classic bounded gradient descent (lbfgs) to find maximum likelihood values for the parameters. When priors are employed to identify maximum posterior probability (MaP) values for the parameters, appropriate synthetic observations (pseudocounts ) are added to the data set to generate prior biases.

Software Support

* **Butils:lbfgsb.c.h** : C wrapper for old Fortran to C code ([link](http://www.ece.northwestern.edu/~nocedal/lbfgsb.html)), do not use this directly, use optimizer.
* **Butils:optimizer.h**: C++ optimizer, provides cleaner interface for options and value / derivative calculations. If analytic derivatives are not available, this header provides a simple difference based approximation. Derive from this base class and override CalcValue(). If analytic derivatives are available, overriding one or both of CalcDeravitive(), CalcValueAndDeravitive() will improve performance.

Phase 4: Model Priors and posteriors.

Yeah Yeah yeah…. We’ll get to this in a moment.

Phase 5: Position wise model posteriors and allele distributions.