Professor of Computer Science Dick Lipton from Georgia Tech has a few days ago posted an interesting post on his blog about a new way of solving systems of linear equations over prime fields. Solving linear systems of equations is a very well explored area and a new approach came as a pleasant surprise to me. The author of the proposed method is Prasad Raghavendra from Berekeley. He wrote a draft paper with the algorithm and the analysis in which he proves the soundness and completeness of the randomized algorithm he had proposed. The proof is not very complicated, you need to be familiar with some probabilistic bounds, specifically a generalization of the Bernstein inequality, the Hoeffding inequality, and foundations of finite fields. Solving linear systems of equations is indeed an ancient problem and a new approach is therefore warmly welcomed. As early as 200 BC the Chinese has devised a clever method for solving 2-by-2 systems and there is evidence that they had developed a method essentially equivalent to Gaussian elimination (2000 years before Gauss came up with it!). Then Gabriel Cramer has in 1750 published a famous and simple O(n^3) rule for solving system of linear equations by computing matrix determinants and cofactors.&#65533;Later in 1810, Carl Friedrich Gauss became interested in discovering the orbit of Pallas, the second-largest asteroid of the solar system. His work led him to a system of six linear equations in six unknowns. Gauss invented the method of Gaussian elimination which is still used today. The method consists of performing row-operations on the coefficient matrix to obtain an equivalent system of equations whose coefficient matrix is upper-triangular. This means that the last equation will involve only one unknown and can be easily solved. Substituting that solution into the second-to-last equation, one can then solve for another unknown. Gauss' work was continued by Wilhelm Jordan who convert a given system of linear equations to a triangular system which can be reduced to a diagonal one and then solved directly (Gauss-Jordan method) -- LU decomposition. We now have faster methods for general systems based on discoveries of Volker Strassen and almost linear time algorithms which require systems to have special structure. Most readers probably know methods mentioned above very well, so let us do not spend any time discussing them. So, what is the trick in the new method? Prasad starts with the set of random vectors, V0. We expect that about half of them will satisfy the first linear equation. He then throws away all vectors that do not satisfy the first equation and call the remaining set S1. The idea is successively applied to further equations until we reach the set Sm. Vectors in Sm will then satisfy all linear equations and represent the solutions to the system. The problem with this winnowing-down process is that with high probability the set Si winnows down to zero before all equations in the system are satisfied. An exception is if we start with a set V0 of exponential size, but this is obviously too much. Prasad came up with a brilliant answer that indeed we do not need a large initial V0 (he proved it can be of linear order in the number of variables) and still have a high probability of getting a solution to the complete system of linear equations if the

system is feasible. The trick is to use an amplifier and a property of finite fields. Remember, the method is for systems of equations over finite field and the exploited property limits its usage. Over finite field with characteristics p, the trick is to form Vi (being of the same size as V0) as sums of random p+1 vectors from Si. Namely, summing p+1 solutions to the first p equations will give in finite field a (new) solution to the first p equations and possibly solve also (p+1)-th solution. See the draft paper for the details. The new algorithm is easy to implement. Recently I am becoming quite a fan of Clojure, so I decided to implement the algorithm in this popular functional programming language that runs on JVM. I have always been very fond of functional programming languages (examples include Haskell, OCaml, F#, Lisp, ML, Erlang, Scheme etc.) and I am glad functional paradigm is gaining popularity also in "real-world" problems (I do not like this term, but ...). Functional programming is not more difficult than object oriented programming, it is just different. It is a style of programming that emphasizes first-class functions that are pure and is inspired by ideas from lambda calculus. I acknowledge that Matlab would be the most appropriate for this problem and imperative language implementation more understandable but it is more fun, isn't it :) ? {codecitation style="brush: python;"} (ns clj_concurrency.ls-solver) &#65533; (declare n m q N) &#65533; (defn random-ints [X] (conj X (repeatedly n #(rand-int q)))) &#65533; (defn mod-reduce [X] (map #(map (fn [x] (mod x q)) %1) X)) &#65533; (defn satisfy? [Ax Sx bx] (<span style="color:

#941c64;">= (mod (apply + (map * Ax Sx)) q) bx)) &#65533; (defn select [S Ai bi] (def T #{}) (doseq [Sx S :when (satisfy? Ai Sx bi)] (def T (conj T Sx))) (into #{} T)) &#65533; (defn recombine [T] (def R #{}) (dotimes [_ N] (let [Q (take (+ q 1) (shuffle T))] (def R (conj R (reduce #(map + %1 %2) Q))))) (into #{} R)) &#65533; (defn solve [A b] (def S (<span style="color:

#941c64;">apply concat (map random-ints (repeat N [])))) (dotimes [i m] (let [T (select S (nth A i) (nth b i))] (when (empty? T) (throw (IllegalArgumentException. "System infeasible, T."))) (def S (recombine T)))) (if (empty? S) (throw (IllegalArgumentException. "System infeasible, S."))) (do (println "--------") (println "Solution" (distinct (mod-reduce S))))) &#65533; <p style="margin:

0px; color: #4c8f75;">; toy example 1 (def n 3) ;number of variables (def m 3) ;number of equations (def q 3) ;finite field characteristics (def N (* 30 n)) ;sampling size (solve [[2 1 1] [1 1 1] [1 2 1]] [1 0 0]) ;--> (1 0 2) &#65533; ; toy example 2 (def q 5) (def m 3) (def n 3) (def N (* 30 n)) (solve [[1 1 1] [2 3 2] [1 3 4]] [1 4 4]) ;--> (1 2 3) {/codecitation}