Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and
Electrical Engineering
Department of Computer and Information Science

Master Thesis

Linear programming on Cell/BE

by

Åsmund Eldhuset

Supervisor: Dr.Ing. Lasse Natvig
Co-supervisor: Dr. Anne C. Elster

Trondheim, June 1, 2009

Abstract

(TODO:)

Acknowledgements

(TODO: Natvig, Elster, Mujahed, Chris, Henrik Andersson, Marielle Christiansen)

Contents

List of Figures

List of Tables

List of Symbols and Abbreviations

Abbreviation	Description	Definition
Cell BE	Cell Broadband Engine	page 20
ILP	Integer linear programming	page 4
ILP	Instruction-level parallelism	page 21
LP	Linear programming	page 3
LS	Local Store	page 21
MFC	Memory Flow Controller	page 21
PPE	PowerPC Processor Element	page 21
PPU	PowerPC Processor Unit	page 21
SPE	Synergistic Processor Element	page 21
SPU	Synergistic Processor Unit	page 21

Listings

Chapter 1

Introduction

(TODO:)

Task description (Natvig: This is the task description I entered in DAIM. Should probably be changed a bit...) The aim of the project is to implement a parallel linear solver for large sparse problems on the Cell BE using the Simplex method. Interior point methods may also be investigated.

(TODO: A paragraph about Miriam)

Chapter 2

Background

(TODO: Chapter introduction)

2.1 Linear programming

(ITP: Section introductions)

This section is primarily based on [25], [5](TODO: and [11] if we write about artificial variables).

2.1.1 Problem formulation. Standard and slack forms

The term *linear programming* (LP) refers to a type of optimisation problems in which one seeks to maximise or minimise the value of a linear function of a set of variables[1]. The values of the variables are constrained by a set of linear equations and/or inequalities. Linear programming is a fairly general problem type, and many important problems can be cast as LP problems — for instance, shortest path problems and maximum flow problems (see [5]). However, the true virtue of linear programming stems from its ability to model a vast range of optimisation problems for which specialised algorithms do not exist, including many situations from economics and industry processes.

The following framed text is an example of a simple linear programming problem. We will use this example throughout this section to illustrate how the linear programming algorithms work.

[1]Hence, LP is not (as the name would seem to suggest) a programming technique. The name originated in the 1940s, when "program" referred to military supply plans and schedules(TODO: citation).

— Example —

A company owns a factory that makes two kinds of products based on two different raw materials. (Natvig: This example just uses some random numbers; I will construct an example, probably using only integers, that can be solved neatly in a few iterations.) The profit the company makes per unit of product A is \$30, and the profit of product B is \$20. Producing one unit of A requires 1 unit of raw material R and 1 unit of raw material S; one unit of B requires 2 units of R and 1 unit of S. The company possesses 40 units of R and 50 units of S. We make the simplifying assumptions that all prices are constant and cannot be affected by the company, and that the company is capable of selling everything it produces. The company's goal is to maximise the profit, which can be described as $30x_1 + 20x_2$, where x_1 is the number of units of product A and x_2 is the number of units of product B. The following constraints are in effect:

- $x_1 + x_2 \leq 40$ (the production of A and B cannot consume more units of raw material R than the company possesses)
- $2x_1 + x_2 \leq 50$ (same for raw material S)
- $x_1, x_2 \geq 0$ (the company cannot produce negative amounts of its products)

Note that in regular LP problems, one cannot restrict the variables to be integers — in fact, adding this requirement produces a new kind of problem known as *integer linear programming* (ILP), which is NP-hard[2]. It is also, in general, a requirement that all variables are nonnegative. This is often the case in real-world problems that deal with physical quantities, but problems involving variables that may be negative as well as positive can still be modeled by rewriting each original variable as a difference of two nonnegative variables.

The function to be optimised is called the *objective function*. In the real world situation that gives rise to an optimisation problem, the function may contain a constant term, but it can be removed since that will affect all possible solutions in the same way. The objective function can then be written as $\zeta = c_1x_1 + c_2x_2 + \ldots + c_nx_n = \sum_{j=1}^{n} c_jx_j$, where the c_j are constants. The variables in the objective function are often called *decision variables*, since our task is not only to find the optimal value of the objective function, but also which variable values that yield

[2]NP-hardness is a term from complexity theory, which deals with the relative difficulties of solving different kinds of problems. The only known algorithms for solving NP-hard problems require an amount of time that is exponential in the size of the problem, which renders those algorithms useless for many real life problem sizes. For further reading on complexity theory, consult [7].

this function value. Throughout this report, we will consistently use n to refer to the number of decision variables and m to refer to the number of equations and/or inequalities. The variables will typically be labelled x_1 through x_n.

Standard form An LP problem is commonly called a *linear program*. The equations and inequalities that (together with the objective function) constitute an linear program may be represented in different forms. We shall first consider the *standard form*, in which only less-than-or-equal-to inequalities with all variables on the left hand side are allowed. (TODO: Why are not less-than allowed?) A problem containing equalities of the form $a_{i1}x_1 + \ldots + a_{in}x_n = b_i$ may be rewritten by splitting each equality into two inequalities: $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$ and $-a_{i1}x_1 - \ldots - a_{in}x_n \leq -b_i$. Also, the goal must be to maximise the objective function — if the original problem is to minimize ζ, we let our objective function be $-\zeta$. A linear program in standard form can be expressed as follows:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.1}$$

with respect to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i, \quad \text{for } i = 1, \ldots, m. \tag{2.2}$$

$$x_1, \ldots, x_n \geq 0 \tag{2.3}$$

Slack form The other common representation is *slack form*, which only allows a set of equations (and a nonnegativity constraint for each variable). A slack form program should be produced by rewriting a standard form program. An inequality of the form $a_{i1}x_i + \ldots + a_{in}x_n \leq b_i$ is converted to an equation by adding a *slack variable* w_i. Together with the condition that $w_i \geq 0$, the equation $a_{i1}x_1 + \ldots + a_{in}x_n + w_i = b_i$ is equivalent to the original inequality (whose difference, or "slack", between the left and right hand sides is represented by w_i). When the program is constructed in this manner, each slack variable only appears in excactly one equation, which is an important property that will be utilised later. A linear program in slack form can be expressed as follows:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.4}$$

with respect to

$$w_i = b_i - \sum_{j=1}^{n} a_{ij} x_j, \quad \text{for } i = 1, \ldots, m. \tag{2.5}$$

$$x_1, \ldots, x_n, w_1, \ldots, w_m \geq 0 \tag{2.6}$$

— Example —

In standard form, our example is expressed as

Maximise

$$\zeta = 30x_1 + 20x_2$$

with respect to

$$x_1 + x_2 \leq 40$$
$$2x_1 + x_2 \leq 50$$
$$x_1, x_2 \geq 0$$

In slack form, it becomes

Maximise

$$\zeta = 30x_1 + 20x_2$$

with respect to

$$w_1 = 40 - x_1 - x_2$$
$$w_2 = 50 - 2x_1 - x_2$$
$$x_1, x_2, w_1, w_2 \geq 0$$

A proposed solution (that is, a specification of a value for each variable) of a linear program in slack form is called:

Feasible if it does not violate any constraints

Infeasible if it violates any constraints

Basic if it consists of setting all variables except the slack variables to zero

Optimal if it is feasible and no other feasible solutions yield a higher value for the objective function

(TODO: Move this paragraph to next section? Natvig says "Ta med etter behov".) The *linear programming theorem* (TODO: Is it actually called this? Find something to cite) states that the optimal solution of a linear program, if it exists, occurs when at least m variables are set to zero. (TODO: Combinatorics. Mention cycling here?)

2.1.2 The simplex method

The *simplex method*[3], developed by George Dantzig[6], was the first systematic approach for solving linear programs. It requires the linear program to be in slack form. The initial coefficients and constants are written down in a *tableau* that will change as the method progresses. The nonnegativity constraints are not represented anywhere, but are implicitly maintained by the method. Because the equations will undergo extensive rewriting, it will be convenient to not distinguish the slack variables from the other variables, so we will relabel w_i to x_{n+i-1} for $i = 1, \ldots, m$. Thus, the total number of variables is $n + m$. Furthermore, we will use overlines over the coefficients in the tableau to denote their *current* value (which will change in each iteration of the simplex method), and the indices of the coefficients will refer to the coefficients' position within the tableau — for instance, $-\overline{a}_{ij}$ is located in row i, column j. We also introduce a constant term $\overline{\zeta}$ (initially zero) in the objective function, which will help us keep track of the best function value we have found so far. The topmost row and leftmost column are not really a part of the tableau; they are simply headers — the topmost row shows which variables correspond to which columns, and the leftmost column shows the slack variables for each row. The first actual tableau row (below the double line) contains the objective function coefficients $\overline{c}_j$; the first actual tableau column (to the right of the double line) contains the $\overline{b}_i$ constants; and the rest of the tableau contains the negatives of the coefficients from the equations: $-\overline{a}_{ij}$. Initially, $\overline{c}_j = c_j$, $\overline{b}_i = b_i$, and $\overline{a}_{ij} = a_{ij}$. For instance, with $n = 3$ and $m = 3$, the initial tableau will look like this:

		x_1	x_2	x_3
ζ	0	c_1	c_2	c_3
x_4	b_1	$-a_{11}$	$-a_{12}$	$-a_{13}$
x_5	b_2	$-a_{21}$	$-a_{22}$	$-a_{23}$
x_6	b_3	$-a_{31}$	$-a_{32}$	$-a_{33}$

Note that this is essentially just a tabular version of the standard form — for instance, the last row is interpreted as the equation $x_6 = \overline{b}_3 - \overline{a}_{31}x_1 - \overline{a}_{32}x_2 - \overline{a}_{33}x_3$.

[3]The reason for not calling it the "simplex *algorithm*" is that there exist several versions of the method, and that the general method formulation is somewhat underspecified because it does not say how to choose the pivot elements.

— Example —

In tableau form, our example becomes

		x_1	x_2
ζ	0	30	20
x_3	40	-1	-1
x_4	50	-2	-1

Note that w_1 and w_2 have been renamed to x_3 and x_4, respectively.

The variables are partitioned into two sets. The variables in the leftmost column (at the left side of the equations) are referred to as the *basic variables*, and the variables inside the tableau are called *nonbasic variables*. At any stage of the method, the set of the indices of the basic variables is denoted $\mathcal{B}$, and the set of nonbasic indices is denoted $\mathcal{N}$. Initially, $\mathcal{N} = \{1, \ldots, n\}$, and $\mathcal{B} = \{n+1, \ldots, n+m\}$. The sizes of the basic and nonbasic sets are constant, with $|\mathcal{B}| = m$ and $|\mathcal{N}| = n$. The tableau will generally look like this (if, for instance, $m = 3$ and $n = 3$):

		$\cdots$	$x_{j\in\mathcal{N}}$	$\cdots$
ζ	$\overline{\zeta}$	$\overline{c}_1$	$\overline{c}_2$	$\overline{c}_3$
$\vdots$	$\overline{b}_1$	$-\overline{a}_{11}$	$-\overline{a}_{12}$	$-\overline{a}_{13}$
$x_{i\in\mathcal{B}}$	$\overline{b}_2$	$-\overline{a}_{21}$	$-\overline{a}_{22}$	$-\overline{a}_{23}$
$\vdots$	$\overline{b}_3$	$-\overline{a}_{31}$	$-\overline{a}_{32}$	$-\overline{a}_{33}$

For now, let us assume that the solution that is obtained by setting all nonbasic variables to zero is feasible (which is the case only if all of the b_i are nonnegative); we will remove this restriction later. This trivial solution will provide a lower bound for the value of the objective function (namely, the constant term). We will now select one nonbasic variable x_j and consider what happens if we increase its value (since all nonbasic variables are currently zero, we cannot decrease any of them). Since our goal is to maximise the objective function, we should select a variable whose coefficient c_j in the objective function is positive. If no such variables exist, we cannot increase the objective function value further, and the current solution (the one obtained by setting all nonbasic variables to zero, so that $\zeta = \overline{\zeta}$) is optimal — we can be certain of this since linear functions do not have local maxima.

It seems reasonable to select the variable with the greatest coefficient. Let us say that this variable is located in column e. For notational convenience, we let $x_{\mathrm{row}\ i}$ denote the basic variable that is located in row i, and we let $x_{\mathrm{col}\ j}$ denote the nonbasic variable in column j. Then, our variable is labelled $x_{\mathrm{col}\ e}$. How far can we increase this variable? Recall that each line in the tableau expresses one

basic variable as a function of all the nonbasic variables; hence we can increase $x_{\text{col } e}$ until one of the basic variables becomes zero. Let us look at row i, which is now reduced to $x_{\text{row } i} = \overline{b}_i - \overline{a}_{ie} x_{\text{col } e}$ since all nonbasic variables except $x_{\text{col } e}$ are zero. If $\overline{a}_{ie}$ is positive, the value of $x_{\text{row } i}$ will decrease as $x_{\text{col } e}$ increases, so the largest allowable increase is limited by $\overline{b}_i$. Thus, by setting $x_{\text{col } e} = \frac{\overline{b}_i}{\overline{a}_{ie}}$, $x_{\text{row } i}$ becomes zero. However, other equations may impose stricter conditions. By looking at all rows where $\overline{a}_{ie}$ is positive, we can determine an l such that $\frac{\overline{b}_l}{\overline{a}_{le}}$ is minimal and set $x_{\text{col } e} = \frac{\overline{b}_l}{\overline{a}_{le}}$. This will cause $x_{\text{row } l}$ to become zero. If all $\overline{a}_{ie}$ are nonpositive, we can increase $x_{\text{col } e}$ indefinitely without any $x_{\text{row } i}$ ever becoming negative, and in that case, we have determined the linear program to be *unbounded*; the method should report this to the user and terminate.

— *Example* —

Recall the tableau:

		x_1	x_2
ζ	0	30	20
x_3	40	-1	-1
x_4	50	-2	-1

Since 30 is the greatest objective function coefficient, we select x_1 to be increased. x_3 becomes zero if $x_1 = \frac{\overline{b}_1}{\overline{a}_{11}} = \frac{40}{1}$, and x_4 becomes zero if $x_1 = \frac{\overline{b}_2}{\overline{a}_{12}} = \frac{50}{2}$. The latter is the most restrictive constraint, so x_4 will become zero when we increase x_1.

The next step, called *pivoting*, is an operation that exchanges a nonbasic variable and a basic variable. The purpose of pivoting is to produce a new situation in which no b_i is negative, so that we can repeat the previous steps all over again and find a new variable whose value we can increase. The nonbasic variable that was selected to be increased, $x_{\text{col } e}$, is called the *entering variable*, since it is about to enter the collection of basic variables. $x_{\text{row } l}$, which becomes zero when $x_{\text{col } e}$ is increased appropriately, is called the *leaving variable*, since it is to be removed from said collection. Keep in mind that since $x_{\text{row } l}$ is a basic variable, it only occurs in one equation, namely

$$x_{\text{row } l} = \overline{b}_l - \sum_{j \in \mathcal{N}} \overline{a}_{lj} x_{\text{col } j}. \tag{2.7}$$

We can eliminate the entering variable from (and introduce the leaving variable into) the set of nonbasic variables by rewriting Equation 2.7:

$$x_{\text{row } l} = \overline{b}_l - \overline{a}_{le} x_{\text{col } e} - \sum_{j \in \mathcal{N} - \{\text{col } e\}} \overline{a}_{lj} x_{\text{col } j} \tag{2.8}$$

$$x_{\text{col } e} = \frac{1}{\overline{a}_{le}} \left(\overline{b}_l - x_{\text{row } l} - \sum_{j \in \mathcal{N} - \{\text{col } e\}} \overline{a}_{lj} x_{\text{col } j} \right). \tag{2.9}$$

Now that we have an expression for $x_{\text{col } e}$, we can substitute it into all of the other equations — this will eliminate $x_{\text{col } e}$ and introduce $x_{\text{row } l}$ into the rest of the tableau. For all $i \in \mathcal{B} - \{\text{row } l\}$, we have:

$$x_{\text{row } i} = \overline{b}_i - \sum_{j \in \mathcal{N}} \overline{a}_{ij} x_{\text{col } j} \tag{2.10}$$

$$= \overline{b}_i - \overline{a}_{ie} x_{\text{col } e} - \sum_{j \in \mathcal{N} - \{\text{col } e\}} \overline{a}_{ij} x_{\text{col } j} \tag{2.11}$$

$$= \overline{b}_i - \frac{\overline{a}_{ie}}{\overline{a}_{le}} \left(\overline{b}_l - x_{\text{row } l} - \sum_{j \in \mathcal{N} - \{\text{col } e\}} \overline{a}_{lj} x_{\text{col } j} \right) - \sum_{j \in \mathcal{N} - \{\text{col } e\}} \overline{a}_{ij} x_{\text{col } j} \tag{2.12}$$

$$= \left(\overline{b}_i - \frac{\overline{a}_{ie}}{\overline{a}_{le}} \overline{b}_l \right) - \frac{\overline{a}_{ie}}{\overline{a}_{le}} x_{\text{row } l} - \sum_{j \in \mathcal{N} - \{\text{col } e\}} \left(\overline{a}_{ij} - \frac{\overline{a}_{ie}}{\overline{a}_{le}} \overline{a}_{lj} \right) x_{\text{col } j}. \tag{2.13}$$

A similar result will be achieved for the expression for the objective function. Although it might look complicated, it amounts to subtracting $\frac{\overline{a}_{ie}}{\overline{a}_{le}}$ times the tableau row for $x_{\text{row } l}$ from all other tableau rows (including the objective function row), and then (TODO:)

Equation 2.8 is the new form of the tableau row that originally corresponded to the basic variable $x_{\text{row } l}$. The new row, which corresponds to $x_{\text{col } e}$, can be easily obtained from the old one by dividing the row by $\overline{a}_{le}$ and setting the coefficient of what is now $x_{\text{row } l}$ to $\frac{1}{\overline{a}_{le}}$.

Finally, we remove row l from $\mathcal{B}$ and add it to $\mathcal{N}$, and remove col e from $\mathcal{N}$ and add it to $\mathcal{B}$.

and adding appropriate multiples of it to each of the other equations: (TODO: Complete this) This step is called a *pivot*. After pivoting, we again have a tableau in which all b_i are nonnegative, and the entire process may be repeated. Note, however, that in subsequent iterations, the indices of the leaving and entering variables may no longer correspond to their respective column and row numbers.

A 3×3 tableau will look like this after one pivot with x_2 as the entering variable and x_5 as the entering variable:(TODO:)

		x_1	x_5	x_3
ζ	$\overline{\zeta}$	c_1	c_2	c_3
x_4	$b_1 - \frac{b_2}{a_{22}}$	$\frac{-a_{11}}{a_{22}}$	$\frac{-a_{12}}{a_{22}}$	$\frac{-a_{13}}{a_{22}}$
x_2	$\frac{b_2}{a_{22}}$	$\frac{-a_{21}}{a_{22}}$	$\frac{-a_{22}}{a_{22}}$	$\frac{-a_{23}}{a_{22}}$
x_6	$\frac{b_3}{a_{22}}$	$\frac{-a_{31}}{a_{22}}$	$\frac{-a_{32}}{a_{22}}$	$\frac{-a_{33}}{a_{22}}$

— *Example* —

After one pivot with x_1 as the entering variable and x_4 as the leaving variable, we get the following tableau:

		x_4	x_2
ζ	750	-15	5
x_3	15	0.5	-0.5
x_1	25	-0.5	-0.5

For the next pivot operation, only x_2 can be selected as the entering variable, which causes x_3 to be selected as the leaving variable. After the pivot, the tableau looks like this:

		x_4	x_3
ζ	900	-10	-10
x_2	30	1	-2
x_1	10	-1	1

Since all objective function coefficients are now negative, we have reached an optimal solution with the value $\zeta = \overline{\zeta} = 900$. This solution value is obtained by setting the nonbasic variables (x_3 and x_4) to 0, in which case $x_1 = 10$ and $x_2 = 30$. We can easily verify that these variable values do not violate any constraints, and by substituting the values into the original objective function, we can verify that the optimal value is indeed $\zeta = 30x_1 + 20x_2 = 30 \cdot 10 + 20 \cdot 30 = 900$.

Degeneracy and cycling (TODO: Briefly discuss degenerate pivots.) A tableau is *degenerate* if some of the $\overline{b}_i$ are zero. Degeneracy may cause problems because a pivot on a degenerate row will not cause the objective function value to change, and we will not have gotten any closer to a solution. With severely bad luck, the algorithm may end up cycling through a number of degenerate states. This, however, rarely happens — according to [25], cycling "is so rare that most efficient implementations do not take precautions against it".

As mentioned in footnote 3 on page 7, the general formulation of the simplex method is underspecified because it does not tell how to break ties between

potential entering and leaving variables. There exist rules that guarantee that cycling will not happen; one of them, called *Bland's rule*[25] is to break ties by always selecting the variable with the smallest index. There are $\binom{m+n}{m}$ possible dictionaries — each dictionary is uniquely determined by the set of basic variables, and the order of the variables is unimportant (if the rows and columns of a dictionary are permuted, it is still regarded as the same dictionary, since the same variables will be selected for pivoting). Since each step transforms one dictionary into another, the simplex method is guaranteed to terminate in at most $\binom{m+n}{m}$ steps if precautions are taken against cycling. In practice, however, the method is usually far more efficient, and algorithms that are guaranteed to run in polynomial time are only superior for very large data sets(TODO: citation).

Initially infeasible problems

The method presented so far is capable of solving linear programs whose initial basic solution (the one obtained by setting all nonbasic variables to 0) is feasible. This is the case only if all of the b_i are nonnegative, which we cannot in general assume them to be. If we have one or more negative b_i, we get around this by introducing an *auxiliary problem* which is based on the original problem, is guaranteed to have a basic feasible solution, and whose optimal solution will provide us with a starting point for solving the original problem. The auxiliary problem is created by subtracting a new variable x_0 from the left hand side of each equation of the original problem (which is assumed to be in standard form), and replacing the objective function with simply $\zeta = -x_0$. The purpose of x_0 is that by initially setting it to a sufficiently large value, we can easily satisfy all equations (even those having negative entries in the right hand side). Then, we can try to change variable values (through regular pivoting) and see if it is possible to make x_0 equal to zero, in which case we can remove it from our equations and reinstate the original objective function, thereby having arrived at a problem that is equivalent to the original one. This is the purpose of our new objective function — since x_0, like all other variables, is required to be nonnegative, the goal of optimising $-x_0$ means that we are trying to make x_0 zero. Fortunately, we do not need a new algorithm for this optimisation process; we can use the simplex algorithm as it has been described above. We only need to do one pivot operation before we start that algorithm: since the idea of x_0 is to initially set it to a suitably large value, and since the algorithm requires a nonnegative right hand side, we should make x_0 a basic variable by performing one pivot operation with the row containing the most negative b_i. This will make the entire right hand side nonnegative. Solving the auxiliary problem is called *Phase I*, and solving the resulting problem (with the original objective function) is called *Phase II*. Thus, the full simplex method is a two-phase method (but of course, if the right hand side of the original problem is nonnegative, we can skip

Phase I).

— *Example* —

(TODO:)

Formal algorithm statement

(TODO: Use the `algorithm` package to give a compact description of the simplex method) (TODO: Should ideally be recognisable in the real code; maybe reference the real code here (or the other way around?))

Complexity and numerical instability

(TODO:) (TODO: P and NC) The complexity classes P and NP should be familiar to anyone that has taken an algorithms course: NP is the class of decision problems (problems that are in the form of a yes/no question) where, if the answer is "yes" and we are given a "certificate" that demonstrates the solution, we can validate the solution in time that is polynomial in the size of the input. P is the subset of NP that consists of those decision problems where we can also *find* the solution in polynomial time. The question of whether $P = NP$ remains one of the most important open questions in the field of computer science(TODO: Clay Millenium Prize). [5] gives a good introduction to complexity theory.

Where does LP fit into this picture? The trivial upper bound of $O(\binom{m+n}{m})$ given above for the number of iterations in the simplex method is absolutely horrible: $\binom{m+n}{m} \geq \left(\frac{m+n}{m}\right)^m = \left(1 + \frac{n}{m}\right)^m$, which, if $m = n$, becomes 2^m. Unfortunately, Klee and Minty[16] proved that it is possible to construct arbitrary-size data sets that make the method hit that bound when a certain pivoting rule is used (and no one has succeeded in finding a pivoting rule that can guarantee polynomial time). However, in practice, the algorithm is often surprisingly efficient(TODO: citation). However, Khachiyan[15] discovered an algorithm that is guaranteed to run in polynomial time, and thus proved LP to be in P.

When dealing with parallel programming, another complexity class is also useful: NC, also known as *Nick's Class*. (TODO:)

In some sense, NC captures the notion of what it means for a problem to be "parallelisable". However, it is not an all-encompassing concept — for instance, (TODO: 2^n to $O(n^c)$ would be great even though not NC)

Greenlaw et al.[8] give a thorough presentation of NC and other aspects of parallel complexity. (TODO: citation for LP being NC)

(ITP: Other stuff that should perhaps be added: geometric interpretation; duality)

(TODO: Warm start)

2.1.3 The revised simplex method

The *revised simplex method* (TODO: citation) is essentially just a linear algebra reformulation of the mathematical operations of the standard simplex method. Rather than

The exposition in this section is based on [25] and [9]. Note that all vectors are column vectors unless stated otherwise.

While this may sound even more time consuming, it turns out that a few tricks will remove the need to perform inversions all of the time. Since most real life problems are sparse, the matrix computations can take that into account and save a lot of time compared to the standard simplex method (in which each iteration requires $O(mn)$ arithmetic operations for the pivot operation).

For these reasons, the revised simplex method is almost always preferred over the standard simplex method in practical implementations (see, for instance, our list of available solvers in Section 2.1.7).

We begin with expressing the slack form constraint tableau in matrix notation. An LP problem in slack form (with renaming of the slack variables) looks like the following:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.14}$$

with respect to

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij} x_j, \quad \text{for } i = 1, \ldots, m. \tag{2.15}$$

$$x_1, \ldots, x_{n+m} \geq 0 \tag{2.16}$$

If we let

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & & \\ a_{11} & a_{12} & \cdots & a_{1n} & & 1 & \\ \vdots & \vdots & \ddots & \vdots & & & \ddots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & & & 1 \end{bmatrix} \tag{2.17}$$

$$\mathbf{b} = \begin{bmatrix} b_1 & \cdots & b_m \end{bmatrix}^\top \tag{2.18}$$

$$\mathbf{c} = \begin{bmatrix} c_1 & \cdots & c_n & 0 & \cdots & 0 \end{bmatrix}^\top \quad (m \text{ zeroes at the end}) \tag{2.19}$$

$$\mathbf{x} = \begin{bmatrix} x_1 & \cdots & x_n & x_{n+1} & \cdots & x_{n+m} \end{bmatrix}^\top \tag{2.20}$$

we can express the problem in a very compact manner:

Maximise

$$\zeta = \mathbf{c}\mathbf{x} \tag{2.21}$$

with respect to

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{2.22}$$

$$\mathbf{x} \geq 0 \tag{2.23}$$

In order to be able to handle the pivot operations, we will need to split each of our matrices and vectors into two in order to reflect which entries correspond to basic variables and which ones do not. As before, we let $\mathcal{N}$ be the collection of nonbasic variable indices (initially $\{1, \ldots, n\}$), and $\mathcal{B}$ the collection of basic variable indices (initially $\{n+1, \ldots, n+m\}$). All the basic variables are put in the vector $\mathbf{x}_\mathcal{B}$, and the nonbasic variables are put in $\mathbf{x}_\mathcal{N}$ — the order of the variables within these vectors do not matter, as long as the entries of the other matrices are arranged correspondingly. We split $\mathbf{A}$ into two matrices: an $m \times n$ matrix $\mathbf{N}$, containing all columns from $\mathbf{A}$ that correspond to nonbasic variables (initially, this will be all the columns containing the a_{ij} entries), and $\mathbf{B}$, which is initially an $m \times m$ identity matrix. Similarly, we split $\mathbf{c}$ into one vector $\mathbf{c}_\mathcal{N}$ for the objective function coefficients belonging to nonbasic variables (initially, $\mathbf{c}_\mathcal{N} = \begin{bmatrix} c_1 & \cdots & c_n \end{bmatrix}^\top$) and one vector $\mathbf{c}_\mathcal{B}$ for the coefficients belonging to basic variables (initially an m element zero vector). After each pivot operation, entries of these matrices and vectors will swap *positions* according to how the collections of basic and nonbasic variables have changed, but the *values* themselves will never change during the course of the algorithm. This means that numerical stability may be significantly improved. Note that the "right hand side" vector $\mathbf{b}$ remains a single vector that will never change. Using these "split" matrices and vectors, we can express the problem as

Maximise

$$\zeta = \mathbf{c}_\mathcal{N}^\top \mathbf{x}_\mathcal{N} + \mathbf{c}_\mathcal{B}^\top \mathbf{x}_\mathcal{B} \tag{2.24}$$

with respect to

$$\mathbf{N}\mathbf{x}_\mathcal{N} + \mathbf{B}\mathbf{x}_\mathcal{B} = \mathbf{b} \tag{2.25}$$

$$\mathbf{x} \geq 0 \tag{2.26}$$

During execution of the (standard) simplex method, it is always the case that each basic variable occurs in exactly one equation, and hence each basic variable can be written as a function of the nonbasic variables. Therefore, $\mathbf{B}$ must be invertible, so we can multiply (2.25) by $\mathbf{B}^{-1}$ and rearrange it to get

$$\mathbf{x}_\mathcal{B} = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_\mathcal{N}. \tag{2.27}$$

Combining this with (2.24), we get

$$\begin{aligned}\zeta &= \mathbf{c}_{\mathcal{N}}^{\top}\mathbf{x}_{\mathcal{N}} + \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{x}_{\mathcal{B}} && (2.28)\\ &= \mathbf{c}_{\mathcal{N}}^{\top}\mathbf{x}_{\mathcal{N}} + \mathbf{c}_{\mathcal{B}}^{\top}(\mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_{\mathcal{N}}) && (2.29)\\ &= \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{b} + (\mathbf{c}_{\mathcal{N}}^{\top} - \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{N})\mathbf{x}_{\mathcal{N}}. && (2.30)\end{aligned}$$

This is very interesting, because it provides explicit formulas for the simplex tableau at any time given the current basic/nonbasic variable configuration. From (2.27) (which can be rewritten as $\mathbf{B}^{-1}\mathbf{N}\mathbf{x}_{\mathcal{N}} + \mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}\mathbf{b}$), we see that the current body of the standard simplex tableau (known as $[\overline{a}_{ij}]$(TODO: this must match the preceding subsection)) can be expressed as $\mathbf{B}^{-1}\mathbf{N}$, and the right hand side of the tableau (known as $[\overline{b}_i]$ — this is also the current values of the basic variables) is $\mathbf{B}^{-1}\mathbf{b}$. Similarly, we see from (2.30) that $\mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{b}$ corresponds to the current value of the objective function (obtained by setting $\mathbf{x}_{\mathcal{N}} = 0$), and the current objective function coefficients (also called the *reduced costs*) from the tableau (known as $[\overline{c}_j]$) are $\mathbf{c}_{\mathcal{N}}^{\top} - \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{N}$. Armed with this knowledge, we can formulate the revised simplex method:

(TODO: Phase I and II)

loa 1: The revised simplex method

procedure REVISEDSIMPLEX(m, n, $\mathbf{N}$, $\mathbf{c}_{\mathcal{N}}$, $\mathbf{b}$)
  Let $\mathbf{c}_{\mathcal{B}}$ be an m element zero vector
  Let $\mathbf{B}$ be an $m \times m$ identity matrix
  Let $\mathbf{B}^{-1}$ be an $m \times m$ identity matrix
  $\mathcal{N} \leftarrow \{1, \ldots, n\}$
  $\mathcal{B} \leftarrow \{n+1, \ldots, n+m\}$
  loop
    $\hat{\mathbf{c}}_{\mathcal{N}}^{\top} \leftarrow \mathbf{c}_{\mathcal{N}}^{\top} - \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{N}$ ▷ Compute the reduced costs
    Search $\hat{\mathbf{c}}_{\mathcal{N}}$ for a negative number; let e be its index (the corresponding nonbasic variable is then $x_{\text{col } e}$)
    if no negative number found in $\hat{\mathbf{c}}_{\mathcal{N}}$ **then**
      return $\mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{b}$, $\mathbf{B}^{-1}\mathbf{b}$ ▷ Optimal value and basic variable values
    end if
    Let $\mathbf{N}_e$ be the eth column of $\mathbf{N}$ (the one corresponding to $x_{\text{col } e}$)
    $\hat{\mathbf{a}} \leftarrow \mathbf{B}^{-1}\mathbf{N}_e$ ▷ Compute the tableau coefficients of $x_{\text{col } e}$
    $\hat{\mathbf{b}} \leftarrow \mathbf{B}^{-1}\mathbf{b}$ ▷ Compute the basic variable values
    Let l be a value of i that minimises $t = \frac{\hat{\mathbf{b}}_i}{\hat{\mathbf{a}}_i}$ (only perform this calculation for those $i \in \mathcal{B}$ where $\hat{\mathbf{a}}_i$ is positive)
    if no value is found for l **then**
      return "The problem is unbounded"
    end if
    Exchange the eth column of $\mathbf{N}$ with the lth column of $\mathbf{B}$
    $\mathcal{B} \leftarrow (\mathcal{B} - \{\text{row } l\}) \cup \{\text{col } e\}$
    $\mathcal{N} \leftarrow (\mathcal{N} - \{\text{col } e\}) \cup \{\text{row } l\}$
    Recalculate $\mathbf{B}^{-1}$ from $\mathbf{B}$
  end loop
end procedure

This method, however, would seem problematic in that it seems to require $\mathbf{B}$ to be inverted in every single iteration. However, it turns out that since only one column of $\mathbf{B}$ changes between iterations, the new $\mathbf{B}^{-1}$ can be calculated from the old one by changing one column (this change can be performed by multiplying by a certain sparse matrix)(TODO: a little bit on eta files). This approach is described in greater detail in Section 8.3 of [25].

2.1.4 ASYNPLEX

[9]

2.1.5 Interior point algorithms

(TODO: Mention Karmarkar, either here or under complexity)

2.1.6 Use of LP to solve advanced flow problems

A *flow network* is a graph where a *flow* of some substance (expressed in e.g. litres per second) is associated with each edge. In addition, each edge may have upper and lower bounds (known as *capacities*) on the flow value, and possibly a *cost* that will be incurred per unit of flow that is sent through the edge. The goal may, for instance, be to send as much flow as possible from a designated *source* (producer) node to a designated *sink* (consumer) node, or to send a certain flow as cheaply as possible. Other variations are also possible. If there are no lower bounds and no costs, there exist efficient algorithms for the maximum flow problem, such as the Edmonds-Karp algorithm[5]. In more complex situations, no specialised algorithms exist, but LP comes to the rescue. Cormen et al.[5] give a good overview of how to express a flow problem as an LP problem, which we summarise here (but only for the simple case of a maximum flow problem). There are two variables for each edge, expressing the amount of flow in each direction through that edge. The flow from node u (directly) to node v is denoted by f_{uv}, and it may not increase above the edge capacity c_{uv} (which may be different in each direction). The following constraints apply:

- The flow in one direction is the negative of the flow in the opposite direction: $f_{uv} = -f_{vu}$, for all u, v.
- The flow through an edge may not exceed the capacity for that edge in that direction: $f_{uv} \leq c_{uv}$, for all u, v.
- Except for the source s and the sink t, all flow entering a node must also exit the node. Due to the "negative flow" convention, this is equivalent to requiring that the flow out of a node is zero: $\sum_v f_{uv} = 0$ for all u except s and t.

The objective is to maximise the flow out of the source (which, by the rules above, must equal the flow into the sink), which is $\sum_v f_{sv}$.

(TODO: Consult Miriam on this)

2.1.7 Existing LP solvers

ILOG CPLEX

CPLEX, developed by the company ILOG, is the industry standard LP solver(Natvig: Who/what can I cite here?). Being proprietary closed-source software, we cannot examine its inner workings (but they are probably too complex for this

project). While our department does not have a CPLEX license, we can still to some extent compare the answers from our solvers to those that CPLEX gives(TODO: as long as the solver is good enough, I guess the answers will always be *correct* — or?) — sites such as (TODO: citation) provide CPLEX' answers to the `netlib` problem sets, and Miriam has a license that they can use to find the answers to their own data sets. (TODO: Something on why Miriam doesn't just use CPLEX rather than bothering with PS3?)

GLPK

Gnu Linear Programming Kit

Unfortunately, the code base is extremely large, comprising more than (TODO:) lines of C code distributed across nearly 100 files. While only a handful of these files contain functionality that is directly related to the simplex method, reverse engineering it still would be a daunting task — especially given that their coding conventions apparently calls for very short variable names.

GLPK is released by its authors under version 3 of the GNU General Public License.

Xpress

`http://www.dashoptimization.com/home//products/products_optimizer.html`

OOPS

`http://www.maths.ed.ac.uk/~gondzio/parallel/solver.html`

CLP

COIN-OR Linear Program Solver (`http://www.coin-or.org/Clp/`).

retroLP

As opposed to virtually all other LP solvers, retroLP[27] implements the original simplex method, not the revised method. The former is advantageous for dense problems, which occur in some special applications such as "wavelet decomposition, digital filter design, text categorization, image processing and relaxations of scheduling problems."[28] As compared to GLPK, the code is fairly short and readable — but it still consists of (TODO:) lines.

retroLP is released by its authors under version 2 of the GNU General Public License.

Vanderbei's code

Vanderbei has published a freely available implementation of the revised simplex algorithm as presented in his book[25], at `http://www.princeton.edu/~rvdb/LPbook/`. While it comprises more than 9000 lines, the core parts are fairly short and well separated from the rest of the code (much of which deals with different input formats).

(TODO: Licence?)

2.1.8 Previous parallelisation attempts

ASYNPLEX (Hall, McKinnon)

[9]

Parallelisation of CPLEX' dual simplex method (Bixby, Martin)

[4]

XPRESS (Andersen, Andersen)

[2]

Parallelisation of interior point algorithms (Karypis, Gupta, Kumar)

[14]

retroLP

(TODO: See above.)

Distributed simplex algorithm (Ho, Sundarraj)

[12]

2.2 Cell Broadband Engine

The *Cell Broadband Engine* (Cell BE) is a single chip multiprocessor architecture jointly developed by IBM, Sony and Toshiba. The initial design goals was to create an architecture that would be suitable for the demands of future gaming and multimedia applications (meaning not only high computational power, but also high responsiveness to user interaction and network communications), with a performance of 100 times that of Sony PlayStation 2[13]. Several obstacles to such goals exist; in particular the infamous *brick walls*[3]:

Memory wall (TODO:)

Power wall (TODO:)

ILP wall *Instruction-level parallelism* (ILP) techniques such as pipelines and (TODO:)

2.2.1 Architecture

Overview

The Cell BE consists of one *PowerPC Processor Element* (PPE) and eight *Synergistic Processing Elements* (SPE)

PPE

PowerPC Processor Unit (PPU) Separate register files for fixed-point, floating-point, and vector. 32 SIMD registers.

SPE

Unified register file with 128 128-bit registers
Synergistic Processor Unit (SPU)

Memory bus and DMA controller

Base addresses (both in local storage and in system memory (TODO: correct?)) for all DMA transfers must be aligned on a 16-byte (quadword) border(TODO: term?), and the data to be transferred must be a multiple of 16 bytes. Performance is improved if aligned, whole cache lines (128 bytes(TODO: verify)) are transferred at a time.
Local Store (LS) *Memory Flow Controller* (MFC)
Another method that is available for communication between the cores is

2.2.2 Programming methods

The `vector` data type

Compiler intrinsics

Compiler directives

`__attribute__((aligned(16)))`, `spu_sel`, `__builtin_expect`, `_align_hint`, `malloc`
loop unrolling, function inlining (watch for code size!)

Branch prediction and avoidance Given the deep pipeline of the SPEs(TODO: verify), branch mispredictions are very expensive. A couple of compiler directives are available to let the programmer help the compiler and the SPE:

1. `__builtin_expect(expression, expected)` will evaluate and return `expression` while informing the compiler that the programmer expects the result to be `expected`. This is typically placed in the condition of an **`if/else`**.

2. If the condition of an **`if/else`** is not easily predictable, but the **`if/else`** bodies are very simple, one might be better off by computing both bodies and using a special selection instruction to determine which result will be kept. `spu_sel(a, b, condition)` will return either `a` or `b` depending on the truth value of `condition`. This translates to (TODO: a single) instruction which does not involve branches.

2.2.3 Tools and libraries

(Natvig's comment: Good: which libs are used in the project? Better: Which libs are relevant for the project?)

BlockLib

[1]

Cell Superscalar

(CellSs) [20]

RapidMind

`http://www.rapidmind.net/`

OpenMP for Cell

[26]

MPI for Cell

(TODO: Add citations: Kumar: A Buffered-Mode MPI Implementation for the Cell BE Processor; Krishna: A Synchronous Mode MPI Implementation on the Cell BE Architecture; JulCe) http://sourceforge.net/projects/mpicell/

(TODO: Locate a BLAS library)

Chapter 3

Design

(TODO: Chapter introduction)

3.1 Overall approach

(TODO: Gradual, step by step approach)

3.2 Initial experiments

3.2.1 Arithmetic performance

(Natvig: Should we do this ourselves, or find someone who has already done it?)

All data in registers

Single precision

Double precision

All data in LS

Single precision

Double precision

Double buffering of data from main storage

(Natvig: Maybe we can find out that data transfer takes so much time that the DP performance loss doesn't have too much of an impact?)

Single precision

Double precision

3.3 Standard simplex method

In order to become familiar with programming the Cell BE, we initially implemented a few versions of the standard simplex method (which is best suited for dense problems). (Natvig's comment: This can be justified when we have a task description and "angrepsmåte") As mentioned in Section 4.1, it turned out that it is extremely hard to make the standard simplex method work reliably on even medium-sized data sets

Our Cell implementation is a fairly straightforward parallelisation of the revised simplex method (that is, it is only a parallel formulation of the same algorithm), so it gives the same results as our sequential implementation. Given that the sequential implementation normally yields a wrong answer for problems of nontrivial size, it is not particularily useful for computations, but Miriam (who has invested in a Cell (TODO: Move this information to the introduction)) was still interested in a demonstration of how much impact the data transfers have on performance. Therefore, we provide here a description of our parallelisation strategy, and in Section 5.2.1 we provide detailed timings of some test runs.

3.3.1 PPE version

(TODO: Far from finished) As described in Section 2.2.1, the PPE supports SIMD instructions (also referred to as vector instructions) capable of operating on four single precision floating point values simultaneously. Since the simplex method primarily consists of row operations on the tableau, it is an excellent target for such vectorisation — the only problem is the low arithmetic intensity, which may reduce performance because a lot of data needs to be loaded into the registers, and only a very simple and fast operation is being performed on each element before it is thrown out again.(TODO: How much does this matter, given the fast LS? However, if the data is too large to fit in the LS, things will probably slow down a lot.)

(TODO: Something on why we chose C++?)

3.3.2 SPE version

Our approach is fairly obvious[1]:

[1] After having written the application, we found that [27] essentially uses the same approach, albeit for cluster computers with MPI.(Natvig: I'm trying to express that although it's not difficult to come up with this approach, I *did* do it myself, before finding that paper. Is that something I should do?)

1. The PPE, which initially holds the entire tableau, distributes the tableau rows evenly among the SPEs (TODO: SPE or SPU?), giving each SPE a batch of consecutive rows.

2. The first SPE analyses the objective function to determine the leaving variable and sends the column number to the PPE, which distributes this number to the other SPEs. If no leaving variable was found, the optimal solution has been found, and the SPEs are asked to send their basic variable values to the PPE and terminate. (TODO: which pivot rule?)

3. Each SPE determines the strictest bound (that is imposed by its subset of the rows) on the value of the leaving variable and sends the bound (TODO: and the corresponding row number) to the PPE.

4. The PPE determines which SPE that "wins" and requests this SPE to transfer the pivot row to main memory; afterwards, all the other SPEs are requested to receive this row(TODO: wording). If no SPEs found a finite bound, the problem is unbounded, and the SPEs are asked to terminate.

5. Each SPE performs row operations on its part of the tableau, using the pivot row, and notify the PPE upon completion. Go to step 2.

3.4 Revised simplex method

(TODO:)

3.4.1 Performing the matrix inversion in parallel

The revised simplex method as described in (TODO: reference) must occasionally spend some time reinverting the basis matrix. A simple yet attractive idea is to offload the matrix inversion onto a separate processor, which may then spend all of its time performing inversions. Then, the main processor can spend all of its time on the remaining steps of the method (while occasionally being provided with a reinverted basis matrix from the inversion processor), and one gets the added benefit of the matrix being reinverted more often (which should be good for numerical stability). (TODO: can the inversion itself be parallelised?) Unfortunately, as reported by Ho and Sundarraj[12, Table 2], the inversion consumes less than 20% of the total time of the revised simplex method, and as such, speedups are limited. Furthermore, this approach does not scale to more than two processors. Therefore, we have chosen not to pursue this direction. Note, however, that ASYNPLEX incorporates the same idea of having a separate inversion processor.

3.5 Thoughts on unimplemented features / ideas for future work

(TODO: Stuff we didn't get the time to do...)

3.5.1 Dense interior point

(TODO: Already done, according to Mujahed (or was it only the Cholesky factorisation step? - acquire reference)

3.5.2 Sparse interior point

The most time consuming step of many interior point algorithms is a Cholesky factorisation.(TODO: citation) Monien and Schulze[19] discuss approaches to parallelising this operation for sparse matrices, and one of those methods, called the *multifrontal method*, is elaborated by Schulze[23].

Andersen and Andersen[2] present a parallel shared memory version of the interior point algorithm that is (or was at the time) underlying the Xpress solver (see Section 2.1.7). Yet another parallel interior point algorithm is presented by Karypis et al.[14].

(TODO: Opportunities for implementing this on cell?)

3.5.3 Mixed precision

[17]

3.5.4 Representation of sparse matrices

Sparse matrices and vectors can be represented in numerous ways; Shahnaz et al.[24] give a good review of different storage schemes. Several operations in a linear solver will depend on the choice of such a representation. If one takes care to place the code for each such operation in a separate function, only a modest amount of work will be required to create implementations of several storage schemes (with the added benefit that these implementations can be tested separately, and as long as they work, the entire solver will still work). Then, one can measure how performance is impacted by the choice of storage scheme.

It should be noted that some formats are intended for general matrices, while others make assumptions about the distribution of nonzeroes — the latter category may be risky to use internally in the solver, since one cannot tell in advance what kind of patterns might emerge in the intermediate matrices produced in the course of the algorithm. (TODO: are we sure about this?) Vanderbei's implementation uses the Compressed Column Storage format, also known as the Harwell-Boeing Sparse Matrix Storage Format[24].

3.5.5 Vectorisation

3.5.6 Autotuning

Chapter 4

Implementation and testing

(TODO: Chapter introduction)

4.1 Implementation problems

4.1.1 Straightforward simplex implementation

Our initial plan was to begin with something we thought to be fairly straightforward and then gradually proceed towards harder problems, along these lines:

1. Implement the standard simplex method on a sequential machine.
2. Parallelise the standard simplex method on Cell (if the Cell turns out to be very hard to program, we could first parallelise it on a regular multicore machine using e.g. OpenMP (see `http://openmp.org/wp/`) to make sure our parallelisation approach is correct).
3. Implement the revised simplex method on a sequential machine.
4. Parallelise the revised simplex method on Cell.
5. Investigate interior point methods and implement them if time permits.

Steps 1 and 2 initially seemed to have been as simple as we had assumed them to be (step 1 was based on the descriptions and pseudocode from [5] and [25]), and the Cell parallelisation went well. These implementations are listed in Appendix A.1. Unfortunately, (TODO:)

Our beliefs were reinforced by the fact that well-known works such as [5] and [25] make no mention of the standard simplex method being particularily *unstable* (they only say that other methods are being used in practice because they are more *efficient*). Also, [21] provided an implementation of the standard simplex method — but when we actually tried it, it turned out to run into the

same kinds of stability problems as our code (TODO: Make a section detailing experiments on this). In the third edition[22], it has been replaced by an implementation of the revised simplex method.

We succeeded in finding an implementation of the standard simplex method that seemed to work well, called retroLP[28]. However, the code base was quite large, and (TODO:)

We eventually resigned and contacted a group of mathematicians with which Lasse is acquainted, asking them for help on how to make the standard simplex method work stably[10]. (TODO:)

Most of the books we have consulted on the subject of linear programming simply give the standard theoretical presentation and completely neglects to mention the practical implementation difficulties — the author of this report would very much have liked a book that is detailing what one needs to do in order to make the simplex method stable. The closest we have come to this is the splendid book by Maros[18]. (TODO:)

MPS parser The `netlib` data sets are stored in a file format called *MPS* (Mathematical Programming System). The format hails from the punch card age; as such, it is fairly arcane (it employs fixed format), but all the simpler to parse. This was fortunate, since we could not find any available parsers, so we had to write our own (. (TODO: Put the source in the appendix) Our parser does not handle all aspects of the format, but(TODO:) Maros[18, Chapter 6] gives a fairly compact presentation of the format.

4.1.2 Numerical stability

(TODO: something on float vs. double?)

In order to prove that the stability problems are not caused by errors in our implementation, we have made our code support use of the *GNU multiple precision arithmetic library* (GMP — see `http://gmplib.org/`), which among other things has a data type for representing arbitrary-size rational numbers exactly. Since the simplex methods only apply the four basic arithmetic operations throughout their operation, all numbers in the tableau will remain rational[1]. Compile the code by running the `buildgmp.sh` script; this will link to GMP and tell our code to use the `mpq_class` data type for all arithmetic operations. When using GMP, the code obviously slows down by a significant factor, but it does produce the right answer for all `netlib` sets.

(TODO: Actually validate this for *all* sets)

[1] Assuming, of course, that they were initially rational — but all data formats for representating of LP problems are based on floating point numbers, which are inherently rational.

4.2 Simplex algorithm

4.3 Test plan

4.3.1 Unit testing

While one might argue that testing an LP solver by running it against a collection of large data sets provides sufficient evidence that the implementation is correct, one will gain even more confidence in the implementation by creating unit tests. Any decent programmer knows how to structure a program by breaking it down into functions, each performing a limited, well-defined part of the overall task. Unit testing, on the other hand, is often neglected, even though it is highly beneficial during development. There is a lot of literature on the subject(TODO: citation), but the basic idea is simple: write code that tests other code. This is fairly straightforward to do as long as the code is partitioned into functions in a reasonable manner. Code should be written to test each nontrivial function for a number of different parameter combinations.

Another important aspect is that unit testing gives *regression testing* for free. If one discovers a bug, one should immediately add a test that demonstrates the bug *before* one fixes the code. That way, one can easily demonstrate that the bug has been fixed, and since this test is now a part of the test suite (*all* of which should be run after each change to *any* code) it will immediately discover the bug if it resurfaces — after all, in large applications bugs in one part of the code can often be triggered .

While some of these considerations are most relevant for software companies, (TODO:)

(TODO: Actually *write* some unit tests...)

4.3.2 Large data sets

(TODO: Something on the `netlib` LP problem set)

4.3.3 Comparison to other implementations

Miriam currently uses the ILOG CPLEX solver, and it would therefore be reasonable to compare the time consumption of our algorithm to those of CPLEX. (TODO: Convert some netlib test sets to cplex format and get Chris to run them) GLPK seems to be the most well-known open source solver, so we might also want to compare our results against it. Of course, since our implementation is based on Vanderbei's code, we will want to measure speedups relative to his implementation. Hall[9] provides relative speedups of ASYNPLEX runs on a few `netlib` sets, with which we can compare our speedups.

Chapter 5

Evaluation

(TODO: Chapter introduction)

5.1 Performance measurements

(TODO: Describe system specifications and how timing was performed)

5.1.1 (TODO: What to measure)

5.1.2 (TODO: How to measure)

5.2 Results

5.2.1 Standard simplex method

5.2.2 Revised simplex method

5.3 Discussion

Chapter 6

Conclusion

(TODO:)

6.1 Experiences

Building an industrial-strength LP solver is a vast amount of work and must only be undertaken with someone who has extensive knowledge of both programming and numerics.

6.2 Future work

Bibliography

[1] M. ÅLIND, M. V. ERIKSSON, AND C. W. KESSLER, *BlockLib: A Skeleton Library for Cell Broadband Engine*, in IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering, New York, NY, USA, 2008, ACM, pp. 7–14. [cited at p. 22]

[2] E. D. ANDERSEN AND K. D. ANDERSEN, *A parallel interior-point algorithm for linear programming on a shared memory machine*, Tech. Rep. 1998008, Université catholique de Louvain, Center for Operations Research and Econometrics (CORE), January 1998. [cited at p. 20, 26]

[3] K. ASANOVÍC, R. BODIK, B. CATANZARO, J. GEBIS, P. HUSBANDS, K. KEUTZER, D. PATTERSON, W. PLISHKER, J. SHALF, S. WILLIAMS, AND K. YELICK, *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences — University of California at Berkeley, December 2006. [cited at p. 20]

[4] R. E. BIXBY AND A. MARTIN, *Parallelizing the Dual Simplex Method*, INFORMS Journal on Computing, 12 (2000), pp. 45–56. [cited at p. 20]

[5] T. H. CORMEN, C. R. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to Algorithms*, McGraw-Hill Science/Engineering/Math, 2nd ed., 2003. [cited at p. 3, 13, 18, 29]

[6] G. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963. [cited at p. 7]

[7] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, 1979. [cited at p. 4]

[8] R. GREENLAW, H. J. HOOVER, AND W. L. RUZZO, *Limits to parallel computation: P-completeness theory*, Oxford University Press, Inc., New York, NY, USA, 1995. [cited at p. 13]

[9] J. A. J. HALL AND K. I. M. MCKINNON, *ASYNPLEX, an asynchronous parallel revised simplex algorithm*, Annals of Operations Research, 81 (1998), pp. 27–50. [cited at p. 14, 17, 20, 31]

[10] HENRIK ANDERSSON (POSTDOC, UNIVERSITY OF BERGEN), *(Private e-mail correspondence)*, April 2009. [cited at p. 30]

[11] F. S. HILLIER AND G. J. LIEBERMAN, *Introduction to Operations Research*, McGraw-Hill Science/Engineering/Math, July 2004. [cited at p. 3]

[12] J. K. HO AND R. P. SUNDARRAJ, *On the efficacy of distributed simplex algorithms for linear programming*, Computational Optimization and Applications, 3 (1994), pp. 349–363. [cited at p. 20, 25]

[13] J. A. KAHLE, M. N. DAY, H. P. HOFSTEE, C. R. JOHNS, T. R. MAEURER, AND D. SHIPPY, *Introduction to the cell multiprocessor*, IBM J. Res. Dev., 49 (2005), pp. 589–604. [cited at p. 20]

[14] G. KARYPIS, A. GUPTA, AND V. KUMAR, *A parallel formulation of interior point algorithms*, in Supercomputing '94: Proceedings of the 1994 ACM/IEEE conference on Supercomputing, New York, NY, USA, 1994, ACM, pp. 204–213. [cited at p. 20, 26]

[15] L. G. KHACHIYAN, *A Polynomial Algorithm in Linear Programming*, Doklady Akademiia Nauk SSSR, 224 (1979), pp. 1093–1096. (English translation in Soviet Mathematics Reports 20:1 (1979), pp. 191–194). [cited at p. 13]

[16] V. KLEE AND G. J. MINTY, *How good is the simplex algorithm?*, in Inequalities, O. Shisha, ed., vol. III, Academic Press, New York, 1972, pp. 159–175. [cited at p. 13]

[17] J. KURZAK AND J. DONGARRA, *Implementation of mixed precision in solving systems of linear equations on the Cell processor: Research Articles*, Concurrency and Computation: Practice & Experience, 19 (2007), pp. 1371–1385. [cited at p. 26]

[18] I. MAROS, *Computational Techniques of the Simplex Method*, Kluwer Academic Publishers, Norwell, MA, USA, 2002. [cited at p. 30]

[19] B. MONIEN AND J. SCHULZE, *Parallel Sparse Cholesky Factorization*. [cited at p. 26]

[20] J. P. PEREZ, P. BELLENS, R. M. BADIA, AND J. LABARTA, *CellSs: making it easier to program the cell broadband engine processor*, IBM Journal of Research and Development, 51 (2007). [cited at p. 22]

[21] W. PRESS, S. TEUKOLSKY, W. VETTERLING, AND B. FLANNERY, *Numerical Recipes in C*, Cambridge University Press, 2nd ed., 1992. [cited at p. 29]

[22] W. H. PRESS, S. A. TEUKOLSKY, W. T. VETTERLING, AND B. P. FLANNERY, *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, 3rd ed., August 2007. [cited at p. 30]

[23] J. SCHULZE, *Parallel Sparse Cholesky Factorization*. [cited at p. 26]

[24] R. SHAHNAZ, A. USMAN, AND I. CHUGHTAI, *Review of Storage Techniques for Sparse Matrices*, in 9th International Multitopic Conference, IEEE INMIC 2005, December 2005, pp. 1–7. [cited at p. 26]

[25] R. J. VANDERBEI, *Linear Programming: Foundations and Extensions*, Springer, 2nd ed., 2001. [cited at p. 3, 11, 12, 14, 17, 20, 29]

[26] H. WEI AND J. YU, *Loading OpenMP to Cell: An Effective Compiler Framework for Heterogeneous Multi-core Chip*, in IWOMP '07: Proceedings of the 3rd international workshop on OpenMP, Berlin, Heidelberg, 2008, Springer-Verlag, pp. 129–133. [cited at p. 22]

[27] G. YARMISH, *A Distributed Implementation of the Simplex Method*, PhD thesis, Polytechnic University, March 2001. [cited at p. 19, 24]

[28] G. YARMISH AND R. V. SLYKE, *retroLP, An Implementation of the Standard Simplex Method*, Tech. Rep. TR-CIS-2001-05, Polytechnic University, August 2001. [cited at p. 19, 30]

Appendices

Appendix A

Code

A.1 Simplex algorithm, first version

matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
#include <vector>
#include <cmath>

#ifdef USE_GMP
  #include <gmpxx.h>
#endif

class Matrix;

class Matrix {
  friend class RowIndexDescriptor;
  friend std::ostream & operator << (std::ostream &, const
     Matrix &);
public:
  Matrix(int, int);
  Matrix(int rows, int cols, TYPE * data);
  Matrix(int, int, bool);
  Matrix(const Matrix &);
  const Matrix & operator = (const Matrix &);
  ~Matrix();
  int getRows() const { return rows; }
  int getCols() const { return cols; }
  TYPE operator () (int r, int c) const;
  TYPE & operator () (int r, int c);
```

```
  Matrix operator + (const Matrix &) const;
  Matrix operator - (const Matrix &) const;
  Matrix operator * (const Matrix &) const;
  Matrix operator * (TYPE) const;
  Matrix transpose() const;
  Matrix invert() const;
  void multiplyRow(int row, TYPE factor);
  void addRows(int sourceRow, int destinationRow, TYPE factor);
  void swapRows(int firstRow, int secondRow);
  void print(const std::vector<int> & basic, const std::vector<
      int> & nonbasic);
private:
  int rows;
  int cols;
  TYPE * data;
};

std::ostream & operator << (std::ostream &, const Matrix &);

inline void incr(TYPE & a, const TYPE & b) {
#ifdef ZEROING_RULE_EPSILON
  a += b;
  if (abs(a) <= EPSILON) {
    a = 0;
  }
#else
#ifdef ZEROING_RULE_RATIO
  TYPE result = a + b;
  if (result == 0 || (abs(a / result) >= RATIO && abs(b /
      result) >= RATIO)) {
    a = 0;
  }
  else {
    a = result;
  }
#else
  a += b;
#endif
#endif
}

#endif
```

matrix.cpp

```
#include "matrix.h"
```

```
#include <cmath>

using namespace std;

Matrix::Matrix(int rows, int cols) {
//  if (rows <= 0 || cols <= 0)
//    throw std::exception();
  this->rows = rows;
  this->cols = cols;
  this->data = new TYPE[rows * cols];
  for (int i = 0;  i < rows * cols; ++i)
    this->data[i] = 0;
}

Matrix::Matrix(int rows, int cols, TYPE * data) {
//  if (rows <= 0 || cols <= 0)
//    throw std::exception();
  this->rows = rows;
  this->cols = cols;
  this->data = new TYPE[rows * cols];
  for (int i = 0;  i < rows * cols; ++i)
    this->data[i] = data[i];
}

Matrix::Matrix(int rows, int cols, bool identity) {
//  if (rows <= 0 || cols <= 0)
//    throw std::exception();
  this->rows = rows;
  this->cols = cols;
  this->data = new TYPE[rows * cols];
  for (int i = 0;  i < rows * cols; ++i)
    this->data[i] = 0;
  if (identity && rows == cols) {
    for (int i = 0; i < rows; ++i) {
      (*this)(i, i) = 1;
    }
  }
}

Matrix::Matrix(const Matrix & source) {
  this->rows = source.rows;
  this->cols = source.cols;
  this->data = new TYPE[source.rows * source.cols];
  for (int i = 0;  i < source.rows * source.cols; ++i)
    this->data[i] = source.data[i];
}
```

```
const Matrix & Matrix::operator = (const Matrix & source) {
  this->rows = source.rows;
  this->cols = source.cols;
  delete this->data;
  this->data = new TYPE[source.rows * source.cols];
  for (int i = 0;  i < source.rows * source.cols; ++i)
    this->data[i] = source.data[i];
  return *this;
}


Matrix::~Matrix() {
  delete [] data;
}

TYPE Matrix::operator () (int r, int c) const {
  if (r < 0 || c < 0 || r >= rows || c >= cols)
    throw "Index out of range";
  return data[r * cols + c];
}

TYPE & Matrix::operator () (int r, int c) {
  if (r < 0 || c < 0 || r >= rows || c >= cols)
    throw "Index out of range";
  return data[r * cols + c];
}

ostream & operator << (ostream & out, const Matrix & matrix) {
  out << "=== " << matrix.rows << " x " << matrix.cols << " ===
      " << endl;
  for (int r = 0; r < matrix.rows; ++r) {
    out << matrix.data[r * matrix.cols];
    for (int c = 1; c < matrix.cols; ++c)
      out << ' ' << matrix.data[r * matrix.cols + c];
    out << endl;
  }
  out << "======" << endl;
  return out;
}

void Matrix::print(const vector<int> & basic, const vector<int>
     & nonbasic) {
  cout << "=== " << rows << " x " << cols << " ===" << endl;
  for (int r = 0; r < rows; ++r) {
    if (r == 0)
      cout << "z = ";
    else
```

```
      cout << "x" << basic[r - 1] << " = ";
    cout << data[r * cols];
    for (int c = 1; c < cols; ++c)
      if (data[r * cols + c] != 0)
        cout << " " << data[r * cols + c] << "x" << nonbasic[c
            - 1];
    cout << endl;
  }
  cout << "======" << endl;
}

Matrix Matrix::operator + (const Matrix & other) const {
  if (rows != other.rows && cols != other.cols)
    throw "Matrix sizes are not equal";
  Matrix result(rows, cols);
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
      result(r, c) = (*this)(r, c) + other(r, c);
    }
  }
  return result;
}

Matrix Matrix::operator - (const Matrix & other) const {
  if (rows != other.rows && cols != other.cols)
    throw "Matrix sizes are not equal";
  Matrix result(rows, cols);
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
      result(r, c) = (*this)(r, c) - other(r, c);
    }
  }
  return result;
}

Matrix Matrix::operator * (const Matrix & other) const {
  if (cols != other.rows)
    throw "Matrices are not compatible";
  Matrix result(rows, other.cols);
  for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < other.cols; ++j) {
      TYPE sum = 0;
      for (int k = 0; k < cols; ++k) {
        sum += (*this)(i, k) * other(k, j);
      }
      if (abs(sum) < 0.00001f)
        sum = 0.0f;
```

```
      result(i, j) = sum;
    }
  }
  return result;
}

Matrix Matrix::operator * (TYPE factor) const {
  Matrix result(rows, cols);
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
      result(r, c) = (*this)(r, c) * factor;
    }
  }
  return result;
}

Matrix Matrix::transpose() const {
  Matrix result(cols, rows);
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
      result(c, r) = (*this)(r, c);
    }
  }
  return result;
}

void Matrix::multiplyRow(int row, TYPE factor) {
  if (factor == 1) return;
  for (int j = 0; j < cols; ++j) {
    (*this)(row, j) *= factor;
  }
}

void Matrix::addRows(int sourceRow, int destinationRow, TYPE
    factor) {
  if (factor == 0) return;
  for (int j = 0; j < cols; ++j) {
    incr((*this)(destinationRow, j), (*this)(sourceRow, j) *
        factor);
  }
}

void Matrix::swapRows(int firstRow, int secondRow) {
  if (firstRow == secondRow) return;
  for (int j = 0; j < cols; ++j) {
    TYPE tmp = (*this)(firstRow, j);
    (*this)(firstRow, j) = (*this)(secondRow, j);
```

```
    (*this)(secondRow, j) = tmp;
  }
}

Matrix Matrix::invert() const {
  if (rows != cols) throw "Non-square matrices cannot be
     inverted";
  Matrix self(*this);
  Matrix inverse(rows, cols, true);
  for (int rc = 0; rc < cols; ++rc) {
    // Locate row with nonzero in this column
    int searchRow = rc;
    while (searchRow < rows && self(searchRow, rc) == 0)
      ++searchRow;
    if (searchRow == rows)
      throw "Matrix is singular";
    // Swap with current row; now the current row has nonzero
       in this column
    self.swapRows(rc, searchRow);
    inverse.swapRows(rc, searchRow);
    TYPE factor = 1 / self(rc, rc);
    self.multiplyRow(rc, factor);
    inverse.multiplyRow(rc, factor);
    for (int r = 0; r < rows; ++r) {
      if (r == rc) continue;
      TYPE factor = -self(r, rc);
      self.addRows(rc, r, factor);
      inverse.addRows(rc, r, factor);
    }
  }
  return inverse;
}
```

TableauSimplex.h

```
#ifndef TALBEAUSIMPLEX_H
#define TALBEAUSIMPLEX_H

#include "matrix.h"
#include <string>
#include <vector>

enum SimplexResult {
  SUBOPTIMAL,
  OPTIMAL,
  UNBOUNDED,
```

```
  CYCLING
};

class TableauSimplex {
public:
  static SimplexResult solve(Matrix & tableau, std::vector<int>
      & basic, std::vector<int> & nonbasic);
  static void pivot(Matrix & tableau, std::vector<int> & basic,
      std::vector<int> & nonbasic, int leaving, int entering);
  static std::string resultToString(SimplexResult result);
};

#endif
```

TableauSimplex.cpp

```
#include "TableauSimplex.h"
#include <cmath>
#include <vector>
#include <climits>
using namespace std;

#define INFINITY 1.0e32f

void TableauSimplex::pivot(Matrix & tableau, std::vector<int> &
    basic, std::vector<int> & nonbasic, int leaving, int
   entering) {
  cout << "Pivoting: " << leaving << " leaves, " << entering <<
      " enters" << endl;
  float xFactor = tableau(leaving, entering);
  int leavingLabel = basic[leaving - 1];
  basic[leaving - 1] = nonbasic[entering - 1];
  nonbasic[entering - 1] = leavingLabel;

/*  cout << "Basic:   ";
  for (unsigned int i = 0; i < basic.size(); ++i)
    cout << " " << basic[i];
  cout << endl << "Nonbasic:";
  for (unsigned int i = 0; i < nonbasic.size(); ++i)
    cout << " " << nonbasic[i];
  cout << endl;*/

  // Cancel out occurrences of the entering variable
  for (int i = 0; i < tableau.getRows(); ++i) {
    if (i == leaving) continue;
    float factor = -tableau(i, entering) / xFactor;
```

```
    float savedColVal = tableau(i, entering);
    tableau.addRows(leaving, i, factor);
    tableau(i, entering) = savedColVal / xFactor;
  }

  tableau.multiplyRow(leaving, -1 / xFactor);
  tableau(leaving, entering) = 1 / xFactor;
}



SimplexResult TableauSimplex::solve(Matrix & tableau, vector<
   int> & basic, vector<int> & nonbasic) {
  int n = tableau.getCols() - 1, m = tableau.getRows() - 1;
  Matrix x(n, 1);

  for (int i = 1; i < n; ++i)
    cout << tableau(0, i) << ' ';
  cout << endl;
  // Find entering variable by searching the objective function
       (row 0) for a positive coefficient (disregard the
      constant in column 0)
  int entering = -1;
  for (int j = 1; j <= n; ++j) {
    //if (tableau(0, j) > 0 && (entering == -1 || tableau(0,
        entering) < tableau(0, j)))// || (tableau(0, entering)
        == tableau(0, j) &&*/ nonbasic[j - 1] < nonbasic[
        entering - 1]))
    if (tableau(0, j) > 0 && (entering == -1 || tableau(0, j) >
         tableau(0, entering) || tableau(0, j) == tableau(0,
        entering) && nonbasic[j - 1] < nonbasic[entering - 1]))
        {
      cout << "Choosing " << j << " over " << entering << " to
         enter; reduced cost is " << tableau(0, j) << endl;
      entering = j;
    }
  }
  if (entering == -1)
    return OPTIMAL;
  cout << "Entering variable: " << nonbasic[entering - 1] << "
      (column " << entering << ")" << endl;

  // Find leaving variable by searching the column of the
      entering variable and determine the strictest bound
  int leaving = -1;
  float largestRatio;
  for (int i = 1; i <= m; ++i) {
```

```
    float ratio;
    if (tableau(i, 0) == 0) {
      if (tableau(i, entering) == 0)
        ratio = 0;
      else if (tableau(i, entering) < 0)
        ratio = INFINITY;
      else
        ratio = -INFINITY;
    }
    else
      ratio = -tableau(i, entering) / tableau(i, 0);
    if (ratio <= 0) continue;
    if (leaving == -1 || ratio > largestRatio || (ratio ==
       largestRatio && basic[i - 1] < basic[leaving - 1])) {
      cout << "Choosing " << i << " over " << leaving << " to
         leave; ratio is " << ratio << endl;
      largestRatio = ratio;
      leaving = i;
    }
  }
  if (leaving == -1)
    return UNBOUNDED;

  cout << "Leaving variable: " << basic[leaving - 1] << " (row
     " << leaving << "); ratio is " << largestRatio << endl;

  pivot(tableau, basic, nonbasic, leaving, entering);


  return SUBOPTIMAL;
}
```

main.cpp

```
#include "matrix.h"
#include "TableauSimplex.h"
#include <cmath>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <fstream>
#include <cstring>

#include "gmpInterop.h"

using namespace std;
```

```
int main(int argc, char * argv[]) {
  int rows, cols;
  bool initiallyFeasible = true;
  bool print = argc >= 3 && strcmp(argv[2], "print") == 0;
  ifstream infile(argv[1]);
  infile >> rows >> cols;
  Matrix A(rows, cols + 1);

  for (int r = 0; r < rows; ++r) {
    for (int c = 1; c < cols; ++c) {
      readNumber(infile, A(r, c));
      if (r > 0) A(r, c) = -A(r, c); // Put the if back when
          doing maximisation
    }
    readNumber(infile, A(r, 0));
    cout << A(r, 0) << endl;
    if (r > 0 && A(r, 0) < 0)
      initiallyFeasible = false;
    A(r, cols) = 1;
  }

  vector<int> basic, nonbasic;
  // Nonbasic variables are labeled 1 .. n
  for (int i = 1; i < cols; ++i)
    nonbasic.push_back(i);
  nonbasic.push_back(0); // Phase I variable
  // Basic variables are labeled n+1 .. n+m
  for (int i = cols; i < cols + rows - 1; ++i)
    basic.push_back(i);

  char cc;
  int itcount = 0;

  // Remember that our A is -A in the article!
  Matrix obj(1, cols); // Saves the original objective function
  if (!initiallyFeasible) {
    cout << "Entering Phase I" << endl;
    for (int c = 0; c < cols; ++c) {
      obj(0, c) = A(0, c);
      A(0, c) = 0;
    }
    A(0, cols) = -1; // The goal is to maximize -x0

    int leaving = 1;
    for (int i = 2; i < rows; ++i) {
      if (A(i, 0) < A(leaving, 0))
```

```
        leaving = i;
    }
    TableauSimplex::pivot(A, basic, nonbasic, leaving, cols);
    if (print) A.print(basic, nonbasic);
    while (TableauSimplex::solve(A, basic, nonbasic) ==
       SUBOPTIMAL) {
      ++itcount;
      if (print) A.print(basic, nonbasic);
      cout << itcount << ": " << A(0, 0) << endl;
    //  cin >> cc;

/*      for (int r = 0; r < A.getRows(); ++r)
        for (int c = 0; c < A.getCols(); ++c)
          if (abs(A(r, c)) < 0.00001)
            A(r, c) = 0;*/
    }
    cout << TableauSimplex::solve(A, basic, nonbasic) << ' ' <<
        itcount << endl;
    cout << "Phase I completed" << endl;
    if (print) A.print(basic, nonbasic);
/*    for (int r = 0; r < A.getRows(); ++r)
      for (int c = 0; c < A.getCols(); ++c)
        if (abs(A(r, c)) < 0.00001)
          A(r, c) = 0;*/
    if (A(0, 0) != 0) {
      cout << "Status: infeasible" << endl;
      return 0;
    }
    if (print) A.print(basic, nonbasic);
  }

  // Locate x0 and
  int x0 = -1;
  for (int i = 0; i < cols; ++i) {
    if (nonbasic[i] == 0) {
      x0 = i + 1;
      nonbasic.erase(nonbasic.begin() + i);
      break;
    }
  }
  Matrix * newTableau;
  if (x0 == -1) {
    for (int j = 0; j < rows - 1; ++j) {
      if (basic[j] == 0) {
        x0 = j + 1;
        basic.erase(basic.begin() + j);
        break;
```

```
      }
    }
    cout << "x0 is not nonbasic, and has value " << A(x0, 0) <<
        endl;
    if (A(x0, 0) != 0)
      return 0;
    newTableau = new Matrix(rows - 1, cols + 1);
    for (int i = /*1*/0; i < rows - 1; ++i) {
      for (int j = 0; j < cols + 1; ++j) {
        (*newTableau)(i, j) = A(i < x0 ? i : i + 1, j);
      }
    }
  }
  else {
    newTableau = new Matrix(rows, cols);
    for (int i = /*1*/0; i < rows; ++i) {
      for (int j = 0; j < cols; ++j) {
        (*newTableau)(i, j) = A(i, j < x0 ? j : j + 1);
      }
    }
  }

  if (!initiallyFeasible) {
    if (print) newTableau->print(basic, nonbasic);
    (*newTableau)(0, 0) = obj(0, 0);//TODO:?
    for (int j = 1; j < cols; ++j)
      if (nonbasic[j - 1] < cols)
        (*newTableau)(0, j) = obj(0, nonbasic[j - 1]);
    for (int i = 1; i < rows; ++i) {
      if (basic[i - 1] < cols) {
        //cout << i << ' ' << obj(0, basic[i - 1]) << endl;
        (*newTableau).addRows(i, 0, obj(0, basic[i - 1]));
      }
    }
//    for (int j = 0; j < newTableau.getCols(); ++j)
//      newTableau(0, j) = -newTableau(0, j);//TODO:??
    //TODO: retain vars from obj
  }

  if (print) newTableau->print(basic, nonbasic);
  cout << "Entering phase II" << endl;
  itcount = 0;
  SimplexResult result;
  while ((result = TableauSimplex::solve(*newTableau, basic,
     nonbasic)) == SUBOPTIMAL) {
    ++itcount;
    if (print) newTableau->print(basic, nonbasic);
```

```
    cout << "iteration " << itcount << ": obj. value is " << (*
       newTableau)(0, 0) << endl;
//    cin >> cc;
  }
  cout << "Status: " << TableauSimplex::resultToString(result)
     << endl;
  if (result == OPTIMAL) {
    for (int i = 1; i < newTableau->getRows(); ++i) {
      if (basic[i - 1] <= newTableau->getCols() && (*newTableau
         )(i, 0) != 0) {
        cout << "x" << basic[i - 1] << ": ";
        printNumber((*newTableau)(i, 0));
        cout << endl;
      }
    }
    cout << "Objective function value: ";
    printNumberFull((*newTableau)(0, 0));
    cout << endl;
  }
  return 0;
}
```

A.2 Utilities

Important note: These parsers are *not* fully compliant with the official MPS and CPLEX file format specifications. They work with the data sets we have used, but have not been thoroughly tested beyond that.

`mps.py` — MPS file format parser

```
from sys import stdin

class Row:
  label = None
  type = None
  values = None
  index = None
  def __init__(self, label, type, index):
    self.label = label
    self.type = type
    self.index = index
    self.values = {}
  def __str__(self):
    return self.label + " (" + self.type + "): " + str(self.
       values)
```

```
lines = []
for line in stdin:
  lines.append(line)

rows = {}
columnLabels = []
columnIndices = {}
i = 0
while i < len(lines):
  line = lines[i]
  i += 1
  if line[0] == ' ':
    pass
  else:
    header = line.strip()
    if header == "ROWS":
      rowIndex = 0
      while lines[i][0] == ' ':
        items = lines[i].split()
        row = Row(items[1].strip(), items[0].strip(), rowIndex)
        if row.type == "N":
          objectiveIndex = rowIndex
        rows[row.label] = row
        rowIndex += 1
        i += 1
        #print row.index, ":", row.label
      tableau = [None] * len(rows)
    elif header == "COLUMNS":
      columnIndex = -1
      while lines[i][0] == ' ':
        items = lines[i].split()
        lim = 2 if len(items) == 5 else 1
        columnLabel = items[0].strip()
        if not columnIndices.has_key(columnLabel):
          columnIndex += 1
          columnLabels.append(columnLabel)
          columnIndices[columnLabel] = columnIndex
        for j in xrange(lim):
          rowLabel = items[1 + j * 2].strip()
          value = float(items[2 + j * 2].strip())
          rows[rowLabel].values[columnLabel] = value
          #print rows[rowLabel].index, ",", columnIndices[
              columnLabel], "=", value
        i += 1
      for j in xrange(len(tableau)):
        tableau[j] = [0] * (len(columnLabels) + 1)
```

```
      for row in rows.values():
        #print "row", row.index, ":", len(row.values)
        for colLabel in row.values:
          tableau[row.index][columnIndices[colLabel]] = row.
             values[colLabel]
    elif header == "RHS":
      while lines[i][0] == ' ':
        items = lines[i].split()
        lim = 2 if len(items) == 5 else 1
        for j in xrange(lim):
          rowLabel = items[1 + j * 2].strip()
          value = float(items[2 + j * 2].strip())
          rowIndex = rows[rowLabel].index
          tableau[rowIndex][-1] = value
          #print "RHS of", rowIndex, "=", value
        i += 1

#print sum([len(r.values) for r in rows.values()])
#for row in tableau:
# tmp = row[-1]
# row[-1] = row[0]
# row[0] = tmp
# print [x for x in row if x != 0]
for row in rows.values():
  tab = tableau[row.index]
  if row.type == "G":
    #print row.index, "is G; multiplying with -1"
    for i in xrange(len(tab)):
      tab[i] = -tab[i]
  elif row.type == "E":
    #print row.index, "is E; creating new row at index ", len(
       tableau)
    tableau.append([-x for x in tab])
#print "objective function is at row", objectiveIndex, ";
   swapping"
tmp = tableau[objectiveIndex]
tableau[objectiveIndex] = tableau[0]
tableau[0] = tmp

ti = 0
while ti < len(tableau):
  nonzero = 0
  for x in tableau[ti]:
    if x != 0:
      nonzero = 1
      break
  if not nonzero:
```

```
    tableau.pop(ti)
    ti -= 1
  ti += 1
#print tableau
#for ti in xrange(len(tableau)):
# tab = tableau[ti]
# newTab = []
# for t in tab[:-1]:
#   newTab.append(t)
#   newTab.append(-t)
# newTab.append(tab[-1])
# tableau[ti] = newTab

#tableau[0] = [-x for x in tableau[0]] #for minimisation?

print len(tableau), len(tableau[0])
for tab in tableau:
  for cell in tab:
    print cell,
  print
sys.exit(0)
print "max: ",
printedAny = 0
for ci in xrange(len(tableau[0]) - 1):
  if tableau[0][ci] != 0:
    if printedAny:
      print " + ",
    printedAny = 1
    print str(tableau[0][ci]) + " x" + str(ci + 1),
print ";"
for tab in tableau[1:]:
  printedAny = 0
  for ai in xrange(len(tab) - 1):
    if tab[ai] != 0:
      if printedAny:
        print " + ",
      printedAny = 1
      print str(tab[ai]) + " x" + str(ai + 1),
  print " <= " + str(tab[-1]) + ";"
for xi in xrange(len(tableau[0]) - 1):
  print "x" + str(xi + 1) + " >= 0;"
```

`cplex.py` — ILOG CPLEX file format parser

```
#!/usr/bin/python
#TODO: "Free" variables may be < 0!
```

```
from sys import stdin, stderr, argv

class Equation:
  comparator = ""
  constant = 0
  values = {}
  name = ""

  def __init__(self, comparator, constant, name):
    self.comparator = comparator
    self.constant = constant
    self.values = {}
    self.name = name



class Bound:
  variable = ""
  lower = 0
  upper = None
  free = False
  fixed = False

  def __init__(self, variable):#, lower, upper):
    self.variable = variable
#   self.lower = lower
#   self.upper = upper



def truncate(name):
  if len(name) <= 8:
    return name
  else:
    return "v" + str(hash(name) % 10000000)



def expand(string, length):
  if len(string) > length:
    raise ValueError("string too long")
  return string + " " * (length - len(string))



class LP:
  pos = 0
```

```
lines = []
variables = {}
equations = []
variableList = []
bounds = []
direction = "max"

def __init__(self):
  lines = []
  variables = {}
  equations = []
  variableList = []

def printMatrix(self):
  eqnCount = 0
  for eq in self.equations:
    if eq.comparator == "=":
      eqnCount += 2
    else:
      eqnCount += 1
  print eqnCount, len(self.variables) + 1
  for eq in self.equations:
    line = [0] * (len(self.variables) + 1)
    line[-1] = eq.constant
    for value in eq.values:
      line[self.variables[value]] = eq.values[value]
    negated = [-x for x in line]
    if eq.comparator == "<=" or eq.comparator == "=" or eq.
       comparator == "obj":
      for x in line:
        print x,
      print
    if eq.comparator == ">=" or eq.comparator == "=":
      for x in negated:
        print x,
      print
  for i in xrange(len(self.variableList)):
    stderr.write(str(i + 1) + ": " + self.variableList[i] + "
       \n")

#WARNING: Truncates names to 10 characters!
def printMPS(self):
  print "NAME          UNKNOWN"
  #print "OBJSENSE"
  #print " " + self.direction.upper()
  print "ROWS"
  for eq in self.equations:
```

```
      if eq.comparator == "=":
        print " E ",
      elif eq.comparator[0] == "<":
        print " L ",
      elif eq.comparator[0] == ">":
        print " G ",
      elif eq.comparator == "obj":
        print " N ",
      else:
        raise NameError("Illegal comparator: " + eq.comparator)
      print expand(truncate(eq.name), 8)
    print "COLUMNS"
    for var in self.variableList:
      for eq in self.equations:
        if eq.values.has_key(var):
          line = expand("    " + truncate(var), 14) + truncate(
              eq.name)
          print expand(line, 24) + str(eq.values[var])
    print "RHS"
    for eq in self.equations:
      if eq.constant != 0:
        print expand("    B         " + truncate(eq.name), 24)
            + str(eq.constant)
    print "BOUNDS"
    for bound in self.bounds:
      if bound.free:
        print " FR BOUND     " + truncate(bound.variable)
      elif bound.fixed:
        print expand(" FX BOUND     " + truncate(bound.variable
            ), 24) + str(bound.upper)
      else:
        if bound.lower != 0:
          print expand(" LO BOUND     " + truncate(bound.
              variable), 24) + str(bound.lower)
        if bound.upper != None:
          print expand(" UP BOUND     " + truncate(bound.
              variable), 24) + str(bound.upper)
    print "ENDATA"

  def printForWebSolver(self):
    print "max: ",
    for eq in self.equations:
      printedAny = 0
      for varName in eq.values:
        if printedAny:
          print "+",
        printedAny = 1
```

```
      print eq.values[varName], varName,
    if eq.comparator != "obj":
      print eq.comparator, eq.constant,
    print ";"
  for v in self.variableList:
    print v, " >= 0;"

def parseObjective(self):
  tokens = self.lines[self.pos]
  self.pos += 1
  self.parseEquation(tokens, 1)

def parseEquation(self, tokens, isObjective):
  if tokens[1] != '+' and tokens[1] != '-':
    tokens.insert(1, '+')
  if isObjective:
    eq = Equation("obj", 0, "OBJ")
  else:
    eq = Equation(tokens[-2], float(tokens[-1]), tokens
        [0][:-1])
  self.equations.append(eq)
  i = 1
  limit = len(tokens) - 1 if isObjective else len(tokens) - 3
  while i < limit:
    if tokens[i] == '-':
      sign = -1
    elif tokens[i] == '+':
      sign = 1
    else:
      print "Illegal sign on line", self.pos, ":", tokens
    if isObjective and self.direction == "max":
      sign *= -1
    try:
      value = float(tokens[i + 1])
      i += 2
    except ValueError:
      value = 1
      i += 1
    name = tokens[i]
    self.addVariable(name)
    eq.values[name] = sign * value
    i += 1

def parseEquations(self):
  while 1:
    tokens = self.lines[self.pos]
    if tokens[0][-1] != ':': break
```

```
    self.pos += 1
    self.parseEquation(tokens, 0)

def addVariable(self, name):
  if not self.variables.has_key(name):
    self.variables[name] = len(self.variables)
    self.variableList.append(name)

def parseBounds(self):
  while 1:
    tokens = self.lines[self.pos]
    if len(tokens) == 1: break
    self.pos += 1
    if len(tokens) == 2 and tokens[1] == "Free":
      bound = Bound(tokens[0])
      bound.free = True
      self.bounds.append(bound)
    elif len(tokens) == 3:
      #TODO: can the eq be turned around?
      #eq = Equation(tokens[1], float(tokens[2]))
      #eq.values[tokens[0]] = 1
      #self.equations.append(eq)
      #self.addVariable(tokens[0])
      bound = Bound(tokens[0])
      if tokens[1][0] == "<":
        bound.upper = float(tokens[2])
      elif tokens[1][0] == ">":
        bound.lower = float(tokens[2])
      elif tokens[1][0] == "=":
        bound.fixed = True
        bound.upper = float(tokens[2])
      else:
        raise NameError("Illegal bound type")
      self.bounds.append(bound)
    elif len(tokens) == 5:
      #eq = Equation(">=", float(tokens[0]))
      #eq.values[tokens[2]] = 1
      #self.equations.append(eq)
      #eq = Equation("<=", float(tokens[4]))
      #eq.values[tokens[2]] = 1
      #self.equations.append(eq)
      #self.addVariable(tokens[2])
      bound = Bound(tokens[2])
      bound.lower = float(tokens[0])
      bound.upper = float(tokens[4])
      self.bounds.append(bound)
    else:
```

```
        print "Unrecognised bounds line:", self.pos, ":",
            tokens


  def parse(self):
    for line in stdin:
      tokens = line.split()
      if len(tokens) == 0 or tokens[0] == '\\': continue
      self.lines.append(tokens)
    self.pos = 0
    while self.pos < len(self.lines):
      if self.lines[self.pos][0] == "Maximize":
        self.direction = "max"
        self.pos += 1
        self.parseObjective()
      elif self.lines[self.pos][0] == "Minimize":
        self.direction = "min"
        self.pos += 1
        self.parseObjective()
      elif self.lines[self.pos][0] == "Subject":
        self.pos += 1
        self.parseEquations()
      elif self.lines[self.pos][0] == "Bounds":
        self.pos += 1
        self.parseBounds()
      else:
        self.pos += 1



lp = LP()
lp.parse()
if len(argv) >= 2 and argv[1] == "web":
  lp.printForWebSolver()
else:
  #lp.printMatrix()
  lp.printMPS()
```

Appendix B

Test sets

B.1 Sample `netlib` test set

This is the `afiro` set, in MPS format(TODO: citation). It contains (TODO: rows, columns and nonzeroes).

```
NAME          AFIRO
ROWS
 E  R09
 E  R10
 L  X05
 L  X21
 E  R12
 E  R13
 L  X17
 L  X18
 L  X19
 L  X20
 E  R19
 E  R20
 L  X27
 L  X44
 E  R22
 E  R23
 L  X40
 L  X41
 L  X42
 L  X43
 L  X45
 L  X46
 L  X47
 L  X48
 L  X49
```

```
 L  X50
 L  X51
 N  COSTCOLUMNS
    X01       X48              .301   R09              -1.
    X01       R10             -1.06   X05               1.
    X02       X21               -1.   R09               1.
    X02       COST              -.4
    X03       X46               -1.   R09               1.
    X04       X50                1.   R10               1.
    X06       X49              .301   R12              -1.
    X06       R13             -1.06   X17               1.
    X07       X49              .313   R12              -1.
    X07       R13             -1.06   X18               1.
    X08       X49              .313   R12              -1.
    X08       R13              -.96   X19               1.
    X09       X49              .326   R12              -1.
    X09       R13              -.86   X20               1.
    X10       X45             2.364   X17              -1.
    X11       X45             2.386   X18              -1.
    X12       X45             2.408   X19              -1.
    X13       X45             2.429   X20              -1.
    X14       X21               1.4   R12               1.
    X14       COST             -.32
    X15       X47               -1.   R12               1.
    X16       X51                1.   R13               1.
    X22       X46              .109   R19              -1.
    X22       R20              -.43   X27               1.
    X23       X44               -1.   R19               1.
    X23       COST              -.6
    X24       X48               -1.   R19               1.
    X25       X45               -1.   R19               1.
    X26       X50                1.   R20               1.
    X28       X47              .109   R22             -.43
    X28       R23                1.   X40               1.
    X29       X47              .108   R22             -.43
    X29       R23                1.   X41               1.
    X30       X47              .108   R22             -.39
    X30       R23                1.   X42               1.
    X31       X47              .107   R22             -.37
    X31       R23                1.   X43               1.
    X32       X45             2.191   X40              -1.
    X33       X45             2.219   X41              -1.
    X34       X45             2.249   X42              -1.
    X35       X45             2.279   X43              -1.
    X36       X44               1.4   R23              -1.
    X36       COST             -.48
```

```
    X37       X49                -1.   R23                 1.
    X38       X51                 1.   R22                 1.
    X39       R23                 1.   COST               10.
RHS
    B         X50               310.   X51               300.
    B         X05                80.   X17                80.
    B         X27               500.   R23                44.
    B         X40               500.
ENDATA
```

B.2 Test sets provided by Miriam AS

These sets are in the ILOG CPLEX format.

dp_0.lp

```
\Problem name: CPLEX solver

Maximize
 obj: v89_49 + zMax557 + zMax558 + id105
Subject To
 CapE50:      v50_49 - RgCapE50  = 0
 OutBal50_49: v50_49 - x536_49  = 0
 CapE51:      v51_49 - RgCapE51  = 0
 OutBal51_49: v51_49 - x538_49  = 0
 CapE52:      v52_49 - RgCapE52  = 0
 OutBal52_49: v52_49 - x540_49 - x542_49  = 0
 CapS59:      v59_49 - RgCapS59  = 0
 InBal59_49:  x536_49 - v59_49  = 0
 OutBal59_49: v59_49 - x548_49 - x550_49  = 0
 CapS60:      v60_49 - RgCapS60  = 0
 InBal60_49:  x550_49 - v60_49 + x544_49  = 0
 OutBal60_49: v60_49 - x552_49  = 0
 CapS61:      v61_49 - RgCapS61  = 0
 InBal61_49:  x538_49 + x540_49 - v61_49  = 0
 OutBal61_49: v61_49 - x554_49  = 0
 CapS62:      v62_49 - RgCapS62  = 0
 InBal62_49:  x542_49 - v62_49  = 0
 OutBal62_49: v62_49 - x556_49  = 0
 CapI89:      v89_49 - RgCapI89  = 0
 InBal89_49:  x548_49 - x544_49 + x554_49 + x556_49 - v89_49 -
    x546_49 = 0
 NetFlowP89:  - v89_49 + vAbs89 >= 0
 NetFlowN89:  v89_49 + vAbs89 >= 0
 CapD53:      v53_49 - RgCapD53  = 0
```

```
 Dem53_49:    d53_49 <= 150
 Dem53:       d53_49 <= 150
 InBal53_49:  x552_49 - v53_49  = 0
 OutBal53_49: - v53_49 + d53_49  = 0
 CapD54:      v54_49 - RgCapD54  = 0
 Dem54_49:    d54_49 <= 150
 Dem54:       d54_49 <= 150
 InBal54_49:  x546_49 - v54_49  = 0
 OutBal54_49: - v54_49 + d54_49  = 0
 Max557:      - d53_49 - d54_49 + zMax557  = 0
 Comp557:     Comp557  = 0
 SMax558:     - v89_49 + zMax558  = 0
Bounds
 0 <= v50_49 <= 200
 0 <= v51_49 <= 200
 0 <= v52_49 <= 200
 0 <= v53_49 <= 200
 0 <= v54_49 <= 200
      zMax557 >= 299.999
      Comp557 Free
      zMax558 >= -0.001
      id105 = 0
 0 <= RgCapE50 <= 200
 0 <= RgCapE51 <= 200
 0 <= RgCapE52 <= 200
 0 <= RgCapS59 <= 120
 0 <= RgCapS60 <= 200
 0 <= RgCapS61 <= 120
 0 <= RgCapS62 <= 120
 0 <= RgCapI89 <= 900000000
 0 <= RgCapD53 <= 200
 0 <= RgCapD54 <= 200
End
```

dp_150.lp

```
\Problem name: CPLEX solver

Maximize
 obj: v89_49 + zMax557 + zMax558 + id105
Subject To
 CapE50:      v50_49 - RgCapE50  = 0
 OutBal50_49: v50_49 - x536_49  = 0
 CapE51:      v51_49 - RgCapE51  = 0
 OutBal51_49: v51_49 - x538_49  = 0
 CapE52:      v52_49 - RgCapE52  = 0
```

```
 OutBal52_49: v52_49 - x540_49 - x542_49  = 0
 CapS59:      v59_49 - RgCapS59  = 0
 InBal59_49:  x536_49 - v59_49  = 0
 OutBal59_49: v59_49 - x548_49 - x550_49  = 0
 CapS60:      v60_49  = 0
 InBal60_49:  x550_49 - v60_49 + x544_49  = 0
 OutBal60_49: v60_49 - x552_49  = 0
 CapS61:      v61_49  = 0
 InBal61_49:  x538_49 + x540_49 - v61_49  = 0
 OutBal61_49: v61_49 - x554_49  = 0
 CapS62:      v62_49  = 0
 InBal62_49:  x542_49 - v62_49  = 0
 OutBal62_49: v62_49 - x556_49  = 0
 CapI89:      v89_49 - RgCapI89  = 0
 InBal89_49:  x548_49 - x544_49 + x554_49 + x556_49 - v89_49 -
    x546_49  = 0
 NetFlowP89:  - v89_49 + vAbs89 >= 0
 NetFlowN89:  v89_49 + vAbs89 >= 0
 CapD53:      v53_49 - RgCapD53  = 0
 Dem53_49:    d53_49 <= 150
 Dem53:       d53_49 <= 150
 InBal53_49:  x552_49 - v53_49  = 0
 OutBal53_49: - v53_49 + d53_49  = 0
 CapD54:      v54_49 - RgCapD54  = 0
 Dem54_49:    d54_49 <= 150
 Dem54:       d54_49 <= 150
 InBal54_49:  x546_49 - v54_49  = 0
 OutBal54_49: - v54_49 + d54_49  = 0
 Max557:      - d53_49 - d54_49 + zMax557  = 0
 Comp557:     Comp557  = 0
 SMax558:     - v89_49 + zMax558  = 0
Bounds
 0 <= v50_49 <= 100
 0 <= v51_49 <= 100
 0 <= v52_49 <= 100
 0 <= v53_49 <= 200
 0 <= v54_49 <= 200
      zMax557 >= 0.999
      Comp557 Free
      zMax558 >= -0.001
      id105 = 0
 0 <= RgCapE50 <= 200
 0 <= RgCapE51 <= 200
 0 <= RgCapE52 <= 200
 0 <= RgCapS59 <= 1
 0 <= RgCapI89 <= 900000000
 0 <= RgCapD53 <= 200
```

```
 0 <= RgCapD54 <= 200
End
```

Appendix C

Schedule

This appendix will obviously be deleted before submission.

Week 20 Study Vanderbei code and obtain a good understanding of how what tricks are required to make revised simplex work in practice; write rough outline of bacground chapter on revised simplex

Either: If the C++ code is too cumbersome to work with:

Week 21 Implement revised simplex (sequentially) in C#, based on Vanderbei

Week 22 Implement ASYNPLEX in C#, based on the above code

Or: If the C++ code is okay to work with:

Week 21 Rewrite Vanderbei's code to become more readable and structured in a way that is more suitable for ASYNPLEX

Week 22 Implement ASYNPLEX in C++, based on the above code

Week 23 Rewrite ASYNPLEX implementation from thread-based C++ or C# code to Cell

Week 24 Run experiments on timing, precision and communication/computation ratio

Week 25 Frenetic report writing

Week 26 — " —

Week 27 — " —; Natvig goes on vacation; I'll try to submit by Friday, July 3

Week 28 Scouting camp (can be dropped if absolutely necessary)

Week 29 Final deadline: Sunday, July 19