(RP) Abstract

Linear programming is a form of mathematical optimisation in which one seeks to optimise a linear function subject to linear constraints on the variables. It is a very versatile tool that has many important applications, one of them being modelling of production and trade in the petroleum industry.

The Cell Broadband Engine, developed by IBM, Sony and Toshiba, is an innovative multicore architecture that has already been proven to have a great potential for high performance computing. However, developing applications for the Cell/B.E. is challenging, particularily due to the low-level memory management that is mandated by the architecture, and because careful optimisation by hand is often required to get the most out of the hardware.

In thesis, we investigate the opportunities for implementing a parallel solver for sparse linear programs on the Cell/B.E.. A parallel version of the standard simplex method is developed, and the ASYNPLEX algorithm by Hall and McKinnon is partially implemented on the Cell/B.E.. We have met substantial challenges when it comes to numerical stability, and this has prevented us from spending sufficient time on Cell/B.E.-specific optimisation and support for large data sets. Our implementations can therefore only be regarded as proofs of concept, but we provide analyses and discussions of several aspects of the implementations, which may guide the future work on this topic.

(RP) Acknowledgements

Contents

List of figures

List of tables

List of algorithms

List of code listings

Chapter 1

(RP) Introduction

This thesis is a part of a larger project that is a cooperation between my advisor — Dr.Ing. Lasse Natvig — and the company Miriam AS. Also involved in that project is Mr. Mujahed Eleyat, whose Ph.D. thesis will be based on our work. Miriam AS develops "Regina" and "Gas", two applications for simulation of production and delivery of oil and gas products through a pipeline network (more information can be found at http://www.miriam.as/). Such situations can be modelled by a *linear program*, which is a linear function of a set of variables along with a set of constraints on the values of other linear functions of those variables. The field of linear programming revolves around the study of algorithms for finding the optimal function values of such systems, and it will be thoroughly presented in the next chapter. There are two main classes of linear programming algorithms: the simplex method (and variations thereof), and interior point methods. Although both are mentioned in the problem description, we only had the time to focus on the former.

Miriam AS has recently invested in a cluster of Playstation 3 machines containing Cell Broadband Engine processors, and is hoping to be able to utilise these to speed up their simulations. The Cell/B.E. is a multicore processor with two different core types: one general-purpose PowerPC core, and eight cores that are specialised for high computational throughput. The computation cores do not have direct access to main memory, but they have a small cache that is controlled manually by the program that is executed on them, and a high-speed bus can be used to efficiently feed the computation cores with data. This sets the Cell/B.E. apart from any other computing platform, and combined with deep pipelines and SIMD instructions, it holds the promise of excellent performance if one has a problem that is suited for that platform and one invests a sufficient amount of time in the programming process.

Unfortunately, it turned out that the field of linear programming is wrought with peril, in the form of numerical instability. Neither we nor our advisor were

prepared for this (we selected this project in the belief that the primary challenge would be to program the Cell/B.E.), and we were not able to fully overcome the problems we encountered. Therefore, we have adjusted our aims to that of producing a code base from which further development may take place, and writing a report that is rich in background material, references and advice that we hope will prove useful to those that are to continue the project.

Outline

(TODO: Make sure this actually fits the final chapter contents!)

In Chapter 2, we give a presentation of the field of linear programming, and we describe the standard and revised simplex methods and a parallel revised simplex method called ASYNPLEX. We also explain the Cell/B.E. architecture and programming model.

In Chapter 3, we describe our initial plans, the progress of our work and the decisions we had to make during the project. We also present our design of a simple parallel standard simplex algorithm for Cell/B.E., and our adaptations of the ASYNPLEX algorithm (which we did not get the time to fully implement). This project ended up very differently from what we had anticipated; we therefore also discuss the challenges we have encountered.

In Chapter 4, we provide several timing analyses of the parallel standard simplex algorithm, in order to learn how the parallelisation, number of cores, and Cell/B.E. features such as vectorisation, affect the performance. We also discuss features we did not get the time to implement, and give several pieces of advice to the researchers that will build upon our work.

Finally, in Chapter 5, we present our conclusions and summarise our suggestions for future work.

Chapter 2

Background

This chapter will give the reader the necessary theoretical background for the main subjects of this thesis: linear programming and the Cell Broadband Engine. We also give some notes on the pthreads API, how to represent sparse vectors, and a formula for the maximal speedup that can be obtained when parallelising a program.

2.1 (RP) Linear programming

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 [7]). 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.

This entire section is primarily based on Vanderbei[38] and Cormen et al.[7].

2.1.1 (RP) Problem formulation. Standard and slack forms

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. 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)

- $2x_1 + x_2 \leq 50$ (similarly 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 $\zeta = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n = \sum_{j=1}^{n} c_j x_j$, 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 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

---

[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 Garey and Johnson[11].

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[3]. 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[4]: $a_{i1}x_1 + \ldots + a_{in}x_n \le b_i$ and $-a_{i1}x_1 - \ldots - a_{in}x_n \le -b_i$. Also, the goal must be to maximise the objective function — if the original problem is to minimize some function f, we let our objective function be $\zeta = -f$. 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

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

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

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 $a_{i1}x_i + \ldots + a_{in}x_n \le b_i$ is converted to an equation by adding a *slack variable* w_i. Together with the condition that $w_i \ge 0$, 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

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

---

[3]Note that strictly-less-than and strictly-greater-than inequalities are never allowed in LP problems, as they could easily cause situations in which it is impossible to achieve optimality — for instance, there is no optimal value for x with respect to $x < 3$; given any value for x that is less than 3, one can always find a number between x and 3.

[4]The drawback of doing this is that it increases the number of equations. See Hillier[20] for another approach, called *artificial variables* — with the drawback that it increases the number of variables.

with respect to

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

$$x_1, \ldots, x_n, w_1, \ldots, w_m \geq 0 \qquad \qquad (2.6)$$

— *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 vector (that is, a specification of a value for each variable) of a linear program is called:

Feasible if it does not violate any constraints;

Infeasible if it violates one or more constraints (however, it is still called a "solution");

Basic if it consists of setting all variables except the slack variables to zero (so that $w_i = b_i$ for all i);

Optimal if it is feasible and no other feasible solutions yield a higher value
for the objective function. An optimal solution vector is not necessarily
unique, although the optimal objective function value obviously is.

2.1.2 (RP) The standard simplex method

The *standard simplex method*, or simply the *simplex method*[5], developed by George
Dantzig[8], was the first systematic approach for solving linear programs. It
requires the linear program to be in slack form. The initial coefficients and con-
stants are written down in a *tableau* that will change as the method progresses.
The nonnegativity constraints are not represented anywhere; rather, they are
implicitly maintained by the method. Because the equations will undergo ex-
tensive rewriting, it will be convenient not to distinguish the slack variables
from the other variables, so we will relabel w_i to x_{n+i} 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 re-
fer to the coefficients' position within the tableau — for instance, $-\bar{a}_{ij}$ is located
in row i, column j. We also introduce a constant term $\bar{\zeta}$ (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 vari-
ables for each row. The first actual tableau row (below the double line) contains
the objective function coefficients $[\bar{c}_j]$ and is numbered as row 0; the first actual
tableau column (to the right of the double line) contains the $[\bar{b}_i]$ constants and
is numbered as column 0; the rest of the tableau contains the negatives of the
coefficients from the equations: $[-\bar{a}_{ij}]$. Initially, $\bar{c}_j = c_j$, $\bar{b}_i = b_i$, and $\bar{a}_{ij} = a_{ij}$.
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 $x_6 = b_3 - a_{31}x_1 - a_{32}x_2 - a_{33}x_3$.

---

[5]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 $|\mathcal{N}| = n$ and $|\mathcal{B}| = m$. The tableau will generally look like this (if, for instance, $m = n = 3$):

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

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, $\bar{\zeta}$). 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 $\zeta = \bar{\zeta}$) 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 (this is known as the *Dantzig criterion* (TODO: verify name); other rules are possible). Let us say that this variable is located in column e. Note that because we will soon start swapping variable positions, the indices of the leaving and entering variables will generally not correspond to their respective row and column num-

bers. For notational convenience, we therefore let $x_{\tilde{i}}$ denote the basic variable that is located in row i, and we let $x_{\hat{j}}$ denote the nonbasic variable in column j. Then, our variable is labelled $x_{\hat{e}}$. 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 $x_{\hat{e}}$ until one of the basic variables becomes zero. Let us look at row i, which is now reduced to $x_{\tilde{i}} = \bar{b}_i - \bar{a}_{ie}x_{\hat{e}}$ since all nonbasic variables except $x_{\hat{e}}$ are zero. If $\bar{a}_{ie}$ is positive, the value of $x_{\tilde{i}}$ will decrease as $x_{\hat{e}}$ increases, so the largest allowable increase is limited by $\bar{b}_i$. Thus, by setting $x_{\hat{e}} = \frac{\bar{b}_i}{\bar{a}_{ie}}$, $x_{\tilde{i}}$ becomes zero. However, other equations may impose stricter conditions. By looking at all rows where $\bar{a}_{ie}$ is positive, we can determine an l such that $\frac{\bar{b}_L}{\bar{a}_{le}}$ is minimal and set $x_{\hat{e}} = \frac{\bar{b}_L}{\bar{a}_{le}}$. This will cause $x_{\tilde{l}}$ to become zero. If all $\bar{a}_{ie}$ are nonpositive, we can increase $x_{\hat{e}}$ indefinitely without any $x_{\tilde{i}}$ 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 $x_1 = \frac{\bar{b}_1}{\bar{a}_{11}} = \frac{40}{1}$, and x_4 becomes zero if $x_1 = \frac{\bar{b}_2}{\bar{a}_{12}} = \frac{50}{2}$. 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 all nonbasic variables are zero (and 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, $x_{\hat{e}}$, is called the *entering variable*, since it is about to enter the collection of basic variables. $x_{\tilde{l}}$, which becomes zero when $x_{\hat{e}}$ is increased appropriately, is called the *leaving variable*, since it is to be removed from said collection. Keep in mind that since $x_{\tilde{l}}$ is a basic variable, it only occurs in one equation, namely

$$x_{\tilde{l}} = \bar{b}_l - \sum_{j \in \mathcal{N}} \bar{a}_{lj}x_{\hat{j}}. \tag{2.7}$$

Note that we have retained all the nonbasic variables, as we want an equation that is valid at all times, not only when almost all nonbasic variables are zero.

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

$$x_{\tilde{l}} = \bar{b}_l - \bar{a}_{le}x_{\hat{e}} - \sum_{j \in \mathcal{N}-\{\hat{e}\}} \bar{a}_{lj}x_{\hat{j}} \tag{2.8}$$

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

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

$$x_{\tilde{i}} = \bar{b}_i - \sum_{j \in \mathcal{N}} \bar{a}_{ij}x_{\hat{j}} \tag{2.10}$$

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

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

$$= \left(\bar{b}_i - \frac{\bar{a}_{ie}}{\bar{a}_{le}}\bar{b}_l \right) + \frac{\bar{a}_{ie}}{\bar{a}_{le}}x_{\tilde{l}} - \sum_{j \in \mathcal{N}-\{\hat{e}\}} \left(\bar{a}_{ij} - \frac{\bar{a}_{ie}}{\bar{a}_{le}}\bar{a}_{lj} \right) x_{\hat{j}}. \tag{2.13}$$

A similar result will be achieved for the expression for the objective function. Although it might look complicated, it amounts to subtracting[6] $\frac{\bar{a}_{ie}}{\bar{a}_{le}}$ times the tableau row l from all other tableau rows i (and adding $\frac{\bar{c}_e}{\bar{a}_{le}}$ times row l to the objective function row), and then setting the tableau entries in column e to $\frac{\bar{a}_{ie}}{\bar{a}_{le}}$ (and to $-\frac{\bar{c}_e}{\bar{a}_{le}}$ in the objective function row). Note that because l was selected such that $\bar{a}_{le}$ was positive and $\frac{\bar{b}_l}{\bar{a}_{le}}$ was minimal, all $\bar{b}_i$ remain nonnegative; and because e was selected such that $\bar{c}_e$ was positive, $\bar{\zeta}$ cannot decrease (it will either retain its old value or increase, depending on whether $\bar{b}_l$ was zero).

(2.9) is the new form of the tableau row that originally corresponded to the basic variable $x_{\tilde{l}}$. The new row, which corresponds to $x_{\hat{e}}$, can be easily obtained from the old one by dividing the row by $\bar{a}_{le}$ and setting the coefficient of what is now $x_{\tilde{l}}$ to $-\frac{1}{\bar{a}_{le}}$.

Finally, we remove $\tilde{l}$ from $\mathcal{B}$ and add it to $\mathcal{N}$, and remove $\hat{e}$ from $\mathcal{N}$ and add it to $\mathcal{B}$, so that the leaving and entering variables swap positions in the new tableau. This completes the pivot operation — we again have a tableau in which all nonbasic variables can be set to zero and all b_i are nonnegative, and the entire process may be repeated.

---

[6]Keeping track of the signs here becomes somewhat cumbersome. Keep in mind that the tableau cell at row i, column j contains $-\bar{a}_{ij}$ (if $i, j \geq 1$).

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: Verify!)

		x_1	x_5	x_3
ζ	$0 + b_2 c_2 / a_{22}$	$c_1 - a_{21} c_2 / a_{22}$	$-c_2 / a_{22}$	$c_3 - a_{23} c_2 / a_{22}$
x_4	$b_1 - b_2 a_{12} / a_{22}$	$-a_{11} + a_{21} a_{12} / a_{22}$	a_{12} / a_{22}	$-a_{13} + a_{23} a_{12} / a_{22}$
x_2	b_2 / a_{22}	$-a_{21} / a_{22}$	$-1 / a_{22}$	$-a_{23} / a_{22}$
x_6	$b_3 - b_2 a_{32} / a_{22}$	$-a_{31} + a_{21} a_{32} / a_{22}$	a_{32} / a_{22}	$-a_{33} + a_{23} a_{32} / a_{22}$

— 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 $\zeta = \bar{\zeta} = 900$. 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 $\zeta = 30x_1 + 20x_2 = 30 \cdot 10 + 20 \cdot 30 = 900$.

2.1.2.1 *(RP)* Degeneracy and cycling

A tableau is *degenerate* if some of the $\bar{b}_i$ 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 Vanderbei[38, p.

32], cycling "is so rare that most efficient implementations do not take precautions against it".

As mentioned in Footnote 5 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*[38, Sec. 3.4] is to break ties by always selecting the variable with the smallest subscript. 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).

2.1.2.2   (RP) Duality

Duality is an interesting property that is exhibited by linear programs, that gives rise to several variations of the standard simplex method.

Given a linear programming problem in standard form:

Maximise

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

with respect to

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

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

its *dual problem* is formed by negating everything and interchanging the roles of rows and columns: the b_i become the objective function coefficients, the c_j become the right hand side, and the positions of the a_{ij} are transposed. Also, the xs are replaced by ys (to avoid confusion with the original problem, since the variables of the dual problems will attain different values in the course of the method). We still want to maximise, but we define the solution of the dual problem to be the negative of the maximal value (this is just a technicality to avoid expressing the problem as a minimisation).

$-$Maximise

$$\xi = -\sum_{i=1}^{m} b_i y_i \tag{2.17}$$

with respect to

$$y_{m+j} = -c_j + \sum_{i=1}^{m} a_{ij} y_i, \;\; \text{for } j = 1, \ldots, n. \tag{2.18}$$

$$y_1, \ldots, y_{m+n} \geq 0 \tag{2.19}$$

This corresponds to negating and transposing the entire tableau. Note that the original problem is referred to as the *primal problem*, and that the dual of the dual problem is the primal problem. There are two highly interesting facts about the dual problem (see [38] for proofs):

The weak duality theorem states that any feasible solution of the dual problem will be greater than any feasible solution of the primal problem.

The strong duality theorem states that the optimal solution of the dual problem equals the optimal solution of the primal problem.

We will not utilise duality extensively, except for the Phase I method discussed below, so we do not give a thorough presentation of it. The concept is very interesting, however, and interested readers should consult Vanderbei[38, Chapter 5], who gives a more in-depth presentation, including an intuitive rationale for the why the dual problem is formed this way.

Duality can be exploited in many ways, one of which is the following: if one has a linear program where the right hand side contains negative numbers, but all objective function coefficients are nonpositive, one can form the dual program (whose right hand side will then contain only nonnegative numbers) and solve that one instead. This approach is called the *dual simplex method*, and it is usually performed without actually transposing the tableau — it just swaps the roles of the basic and nonbasic variables.

2.1.2.3 *(RP)* 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 if and only if all of the b_i are nonnegative, which we cannot in general assume them to be. As mentioned in the preceding section, one can get around this if all the c_j are nonpositive, but this does not generally hold either. 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[7]). 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).

Another Phase I method, the one used by Vanderbei, is to first replace negative terms in the objective function by an arbitrary positive number (e.g. 1) and then run the dual simplex method as described above. The dual method will terminate when the original right hand side only consists of nonnegative numbers, in which case we can reinstate the actual coefficients of the original objective function and proceed with Phase II.

One-phase methods also exist, such as the *parametric self-sual simplex method*, as described in [38, Sec. 7.3].

(If time permits: example)

2.1.2.4   (RP) Formal algorithm statement

In Algorithm 1 on the next page we present the pseudocode for an individual phase of the standard simplex method (with the first approach described in Section 2.1.2.3, the same code can be used for both Phase I and Phase II. The tableau is called T and is zero-indexed; keep in mind that row 0 is the objective function and column 0 contains the constants from the right hand sides of the inequali-

---

[7]Beware that "the right hand side" refers to the b_i, which are on the right hand side of the original equations — but in the tableau, they are on the *left* side.

ties. The current value of the objective function is always in row 0, column 0. We use row major indexing, so $T[2, 3]$ is row 2, column 3. (Natvig's comment: Should ideally be recognisable in the real code; maybe reference the real code here (or the other way around?))

(TODO: Verify)

loa 1: One phase of the standard simplex method using the Dantzig criterion TODO: verify name

```
 1: procedure STANDARDSIMPLEXPHASE(m, n, a[1..m, 1..n], b[1..m], c[1..n])
 2:     T[0, 0] ← 0
 3:     T[i, j] ← −a[i, j] for i = 1 . . . m, j = 1 . . . n
 4:     T[i, 0] ← b[i] for i = 1 . . . m
 5:     T[0, j] ← c[j] for j = 1 . . . n
 6:     N ← {1, . . . , n}
 7:     B ← {n + 1, . . . , n + m}
 8:     loop
 9:         Pick a column number e ≥ 1 such that T[0, e] is positive and maximal
10:         if no e is found then
11:             return T[0, 0] as the optimal solution
12:         end if
13:         Pick a row number l ≥ 1 such that T[l, e] < 0 and − T[l,0]/T[l,e] is minimal
14:         if no l is found then
15:             return "The problem is infeasible" (if this is Phase I) or "The prob-
    lem is unbounded" (if this is Phase II)
16:         end if
17:         p ← −T[l, e]
18:         for i ← 0, m do
19:             if i ≠ l then
20:                 f ← T[i,e]/p
21:                 Add f times row l of T to row i of T
22:                 T[i, e] ← −f
23:             end if
24:         end for
25:         Divide row l of T by p
26:         T[l, e] ← − 1/p
27:     end loop
28: end procedure
```

2.1.2.5 (RP) Complexity and numerical instability

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, and is one of the seven Clay Millennium Prize problems[8]. Most researchers believe that $P \subset NP$, and that the most difficult problems in *NP*, the so-called *NP-complete* (*NPC*) problems, cannot be solved in polynomial time. Cormen et al.[7] give a good introduction to complexity theory.

When dealing with parallel programming, another complexity class is also useful: *NC*, also known as *Nick's Class*. This is the class of all problems that can be solved in $O(\lg^{k_1} n)$ steps (so-called *polylogarithmic time*) using a polynomial ($O(n^{k_2})$) number of processors. Here, k_1 and k_2 are constants. *NC* is a subset of *P*, since any parallel algorithm requiring $f(n)$ steps using $p(n)$ processors can be simulated in $p(n)f(n)$ steps on a sequential computer. Thus, any *NC*-algorithm will require $O(n^{k_2} \lg^{k_1} n)$ steps on a sequential machine, and this is polynomial in n. However, there are problems in *P* which have not yet been proven to be in *NC*, and the most difficult problems among these are called *P-complete* (*PC*) — this is quite analoguous to the *NP/P/NPC* situation.

In some sense, *NC* captures the notion of what it means for a problem to be "parallelisable", while the *P*-complete problems can be said to be "hard to parallelise". However, it is not an all-encompassing concept — a problem may be in *NC* without being efficiently solvable in practice due to a prohibitive processor requirement of the algorithm (for instance $O(n^{10})$ processors) or large constants hidden by the *O*-notation, and a parallel algorithms for *P*-complete problems may still be useful because they might be faster than their sequential counterparts (just not "much faster").

Where does LP fit into this picture? The trivial upper bound of $O(\binom{m+n}{m})$ given in Section 2.1.2.1 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[26] 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). In spite of this, the method is often surprisingly efficient in practice(TODO: citation). In 1979, Khachiyan[25] discovered a different kind of algorithm that is guaranteed to run in polynomial time, and thus he proved LP to be in *P*.[9] However, LP is also *P*-complete, as proved by Dobkin et al.[9]. Still, for the reasons mentioned above, this should not discourage us from

---

[8]http://www.claymath.org/millennium/

[9]Strictly speaking, LP is a computation problem (one in which we seek a numerical answer) rather than a decision problem and thus falls outside of the *NP/P/NC* discussion. However, like many other computation problems, LP easily can be reformulated as a decision problem that can be solved by the same algorithms; see [15, Problem A.4.3] for more references.

seeking parallel versions of LP algorithms.

Greenlaw et al.[15] give a thorough presentation of *NC* and other aspects of parallel complexity, and a more compact survey of the field can be found in Natvig(TODO: citation).

2.1.2.6 (RP) Warmstarting

If one has solved an LP problem and then wishes to solve a very similar problem (one that has been obtained by slightly altering the various coefficients of the original problem), it would seem reasonable to believe that the optimal solution to the original problem would be a great starting point in the search for the optimal solution to the new problem. This turns out to be the case, and the idea is known as *warmstarting*. It normally leads to a great reduction in the time required to solve the new problem, and it is also very easy to implement — the simplex method need not be changed at all; the program must simply be capable of taking a suggested starting solution as input. Note that one might have to run both phases, in case the original solution is not feasible for the new problem. Interested readers may consult Vanderbei[38, Chapter 7] for a more thorough introduction to the subject (which he refers to as *sensitivity analysis*).

Miriam employs Monte Carlo methods[10] that produce a number of random variations of the current state of the oil pipeline network in order to predict what will happen if anything changes. (TODO: more information) This is an important reason that they want to focus on the simplex method rather than interior point methods (Section 2.1.5) — warmstarting is possible for the latter class of methods, but it is much harder to implement. Various approaches to warmstarting interior point methods are described by e.g. Gondzio and Grothey[14] (this is actually a more general approach for quadratic programming), Yildirim and Wright[43], and Benson and Shanno[5].

2.1.3 (RP) 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; however, it is much more numerically stable, for reasons that will be explained. We begin with expressing the slack form constraint tableau in matrix notation — note that all vectors are column vectors unless stated otherwise. An LP problem in slack form (with renaming of the slack variables) looks like the following:

Maximise

$$\zeta = \sum_{j=1}^{n} c_j x_j \tag{2.20}$$

---

[10]TODO: describe Monte Carlo methods

with respect to

$$x_{n+i} = b_i - \sum_{j=1}^{n} a_{ij}x_j, \text{ for } i = 1, \ldots, m. \quad (2.21)$$

$$x_1, \ldots, x_{n+m} \geq 0 \quad (2.22)$$

If we let

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & 1 & & \\ a_{21} & a_{22} & \cdots & a_{2n} & & 1 & \\ \vdots & \vdots & \ddots & \vdots & & & \ddots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & & & & 1 \end{bmatrix} \quad (2.23)$$

$$\mathbf{b} = \begin{bmatrix} b_1 & \cdots & b_m \end{bmatrix}^\top \quad (2.24)$$

$$\mathbf{c} = \begin{bmatrix} c_1 & \cdots & c_n & 0 & \cdots & 0 \end{bmatrix}^\top \ (m \text{ zeroes at the end}) \quad (2.25)$$

$$\mathbf{x} = \begin{bmatrix} x_1 & \cdots & x_n & x_{n+1} & \cdots & x_{n+m} \end{bmatrix}^\top \quad (2.26)$$

we can express the problem in a very compact manner:

Maximise

$$\zeta = \mathbf{c}^\top \mathbf{x} \quad (2.27)$$

with respect to

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad (2.28)$$

$$\mathbf{x} \geq 0 \quad (2.29)$$

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, since the matrices and vectors will not accumulate errors (practical implementations do, however, maintain additional vectors that will accumulate errors; see below). 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

$$\zeta = \mathbf{c}_{\mathcal{N}}^{\top}\mathbf{x}_{\mathcal{N}} + \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{x}_{\mathcal{B}} \qquad (2.30)$$

with respect to

$$\mathbf{N}\mathbf{x}_{\mathcal{N}} + \mathbf{B}\mathbf{x}_{\mathcal{B}} = \mathbf{b} \qquad (2.31)$$
$$\mathbf{x} \geq 0 \qquad (2.32)$$

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 (2.31) by $\mathbf{B}^{-1}$ and rearrange it to get

$$\mathbf{x}_{\mathcal{B}} = \mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_{\mathcal{N}}. \qquad (2.33)$$

Combining this with (2.30), we get

$$\zeta = \mathbf{c}_{\mathcal{N}}^{\top}\mathbf{x}_{\mathcal{N}} + \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{x}_{\mathcal{B}} \qquad (2.34)$$
$$= \mathbf{c}_{\mathcal{N}}^{\top}\mathbf{x}_{\mathcal{N}} + \mathbf{c}_{\mathcal{B}}^{\top}(\mathbf{B}^{-1}\mathbf{b} - \mathbf{B}^{-1}\mathbf{N}\mathbf{x}_{\mathcal{N}}) \qquad (2.35)$$
$$= \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{b} + (\mathbf{c}_{\mathcal{N}}^{\top} - \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{N})\mathbf{x}_{\mathcal{N}}. \qquad (2.36)$$

This is very interesting, because we can use it to acquire explicit formulas for the simplex tableau at any time during the method, given the current basic/nonbasic variable configuration: From (2.33) (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 negative of the current body of the standard simplex tableau (the coefficients that in the previous subsection were known as $[-\bar{a}_{ij}]$) can be expressed as $\mathbf{B}^{-1}\mathbf{N}$.

- The "right hand side" (the leftmost column of the tableau, known as $[\bar{b}_{i}]$ — these are also the current values of the basic variables) is $\mathbf{B}^{-1}\mathbf{b}$.

- Similarly, we see from (2.36) 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 $[\bar{c}_{j}]$) are $\mathbf{c}_{\mathcal{N}}^{\top} - \mathbf{c}_{\mathcal{B}}^{\top}\mathbf{B}^{-1}\mathbf{N}$.

How can we express a pivot operation? It turns out to be exceedingly simple — if the entering variable is $x_{\hat{e}}$ and the leaving variable is $x_{\tilde{l}}$, it is sufficient to swap column e of $\mathbf{N}$ with column l of $\mathbf{B}$. We do not even need to physically move the columns; we can implicitly move them by using permutation lists (based on $\mathbf{B}$ and $\mathbf{N}$) to keep track of which columns are located where. Strangely, Vanderbei[38] does not seem to contain a proof that pivots can be performed in this manner (for the most part, the book is burgeoning with useful proofs). For the sake of completeness, we provide here a (not entirely rigorous) demonstration that at least the first pivot will work. $\mathbf{B}$ was initially an identity matrix, and $\mathbf{N}$ was $[\bar{a}_{ij}]$. We now perform a pivot where $x_{\hat{e}}$ enters and $x_{\tilde{l}}$ leaves — this will cause column e from $\mathbf{N}$ to be swapped with column l from $\mathbf{B}$. Let us assume that the former column does not contain any zeroes (if it does, its inverse will look different); then we have the following (column l of $\mathbf{B}$ and $\mathbf{B}^{-1}$ and column e and row l of $\mathbf{N}$ have been (TODO: spell "emphasized")):

$$
\mathbf{B}^{-1} =
\begin{bmatrix}
1 & & & a_{1e} & & \\
 & \ddots & & \vdots & & \\
 & & & a_{le} & & \\
 & & & \vdots & \ddots & \\
 & & & a_{me} & & 1
\end{bmatrix}^{-1}
=
\begin{bmatrix}
1 & & & -a_{1e}/a_{le} & & \\
 & \ddots & & \vdots & & \\
 & & & 1/a_{le} & & \\
 & & & \vdots & & \ddots \\
 & & & -a_{me}/a_{le} & & 1
\end{bmatrix}
$$

and

$$
\mathbf{B}^{-1}\mathbf{N} =
\begin{bmatrix}
1 & & & -a_{1e}/a_{le} & & \\
 & \ddots & & \vdots & & \\
 & & & 1/a_{le} & & \\
 & & & \vdots & \ddots & \\
 & & & -a_{me}/a_{le} & & 1
\end{bmatrix}
\begin{bmatrix}
a_{11} & \cdots & 0 & \cdots & a_{1n} \\
\vdots & & \vdots & & \vdots \\
a_{l1} & \cdots & 1 & \cdots & a_{ln} \\
\vdots & & \vdots & & \vdots \\
a_{m1} & \cdots & 0 & \cdots & a_{mn}
\end{bmatrix}
$$

$$
=
\begin{bmatrix}
a_{11} - a_{l1}a_{1e}/a_{le} & \cdots & -a_{1e}/a_{le} & \cdots & a_{1n} - a_{ln}a_{1e}/a_{le} \\
\vdots & & \vdots & & \vdots \\
a_{l1}/a_{le} & \cdots & 1/a_{le} & \cdots & a_{ln}/a_{le} \\
\vdots & & \vdots & & \vdots \\
a_{m1} - a_{l1}a_{me}/a_{le} & \cdots & -a_{me}/a_{le} & \cdots & a_{mn} - a_{ln}a_{me}/a_{le}
\end{bmatrix}
$$

Compare this to the tableau on page 11 (where $m = n = 3$ and $e = l = 2$) — its main body is the exact negative of this matrix, as expected. Similar derivations can be carried out for the right hand side and for the objective function coefficients.

Armed with this knowledge, we can formulate the revised simplex method, as shown in Algorithm 2 on the facing page. Note that, like the standard simplex method, it may also require two phases, and it is still necessary to specify a way of selecting the entering variable.

loa 2: One phase of the revised simplex method

1: **procedure** REVISEDSIMPLEXPHASE(m, n, $\mathbf{N}$, $\mathbf{c}_{\mathcal{N}}$, $\mathbf{b}$)
2:      Let $\mathbf{c}_{\mathcal{B}}$ be an m element zero vector
3:      Let $\mathbf{B}$ be an $m \times m$ identity matrix
4:      Let $\mathbf{B}^{-1}$ be an $m \times m$ identity matrix
5:      $\mathcal{N} \leftarrow \{1, \ldots, n\}$
6:      $\mathcal{B} \leftarrow \{n+1, \ldots, n+m\}$
7:      **loop**
8:        $\hat{\mathbf{c}}_{\mathcal{N}}^{\top} \leftarrow \mathbf{c}_{\mathcal{N}}^{\top} - \mathbf{c}_{\mathcal{B}}^{\top} \mathbf{B}^{-1} \mathbf{N}$ $\triangleright$ Compute the reduced costs
9:        Search $\hat{\mathbf{c}}_{\mathcal{N}}$ for a negative number; let e be its index (the corresponding nonbasic variable is then $x_{\hat{e}}$)
10:        **if** no negative number found in $\hat{\mathbf{c}}_{\mathcal{N}}$ **then**
11:          **return** $\mathbf{c}_{\mathcal{B}}^{\top} \mathbf{B}^{-1} \mathbf{b}$, $\mathbf{B}^{-1}\mathbf{b}$ $\triangleright$ Optimal value and basic variable values
12:        **end if**
13:        Let $\mathbf{N}_e$ be the eth column of $\mathbf{N}$ (the one corresponding to $x_{\hat{e}}$)
14:        $\hat{\mathbf{a}} \leftarrow \mathbf{B}^{-1}\mathbf{N}_e$ $\triangleright$ Compute the tableau coefficients of $x_{\hat{e}}$
15:        $\hat{\mathbf{b}} \leftarrow \mathbf{B}^{-1}\mathbf{b}$ $\triangleright$ Compute the basic variable values
16:        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)
17:        **if** no value is found for l **then**
18:          **return** "The problem is unbounded"
19:        **end if**
20:        Exchange the eth column of $\mathbf{N}$ with the lth column of $\mathbf{B}$
21:        $\mathcal{B} \leftarrow (\mathcal{B} - \{\tilde{l}\}) \cup \{\hat{e}\}$
22:        $\mathcal{N} \leftarrow (\mathcal{N} - \{\hat{e}\}) \cup \{\tilde{l}\}$
23:        Recalculate $\mathbf{B}^{-1}$ from $\mathbf{B}$
24:      **end loop**
25: **end procedure**

This method looks 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 in iteration, each $\mathbf{B}^{-1}$ can be calculated from the previous one by changing one column; furthermore, this change can be expressed as a multiplication with a sparse matrix formed in a certain way. A chain of such matrices is called an *eta file*, and this approach is described in greater detail in [38, Section 8.3]. Of course, the longer the eta file gets, the slower the calculation will become, and inaccuracies may accumulate. Therefore, with regular intervals, $\mathbf{B}^{-1}$ should be recomputed from scratch from the current version of $\mathbf{B}$. This will also eliminate the inaccuracies (unless $\mathbf{B}$ is ill-conditioned, in which case one may run into problems). Note that it is possible to update $\hat{\mathbf{b}}$ and $\hat{\mathbf{c}}_{\mathcal{N}}$ in each iteration rather than to recalculate them (this is the approach taken by [38]), but the update calculations are also time consuming.

2.1.4   (RP) ASYNPLEX

As we will describe in Section 3.3, even getting With time becoming scarce, we realised that we most likely would not be able to develop an algorithm of our own for a parallel revised simplex method, and so we started looking for existing algorithms. We did not find many, and the most promising one (in particular because it bears a strong resemblance to the original revised simplex method) is called ASYNPLEX, and was developed by Hall and McKinnon[18]. It is an asynchronous algorithm[11] for message-passing systems, but the authors also describe a shared-memory version of the algorithm. We will now present ASYNPLEX, based on [18]. We did need to make some small changes to the algorithm, and we have described those in (TODO: reference).

Before proceeding, we should mention that on the coarsest level, one can distinguish between two ways of achieving parallelism:

Task parallelism can be achieved when two or more different operations can be performed in parallel.

Data parallelism can be achieved when the same operation is applied to several pieces of data (TODO: reformulate).

The extent to which the different parts of the computation are independent will greatly affect the possibilities for speedup. Computations that can be split into parts that are entirely independent are called *embarrassingly parallel* (see Section 2.3.3 for more information on this), and such computations will benefit greatly from parallelisation (unless the computation is so simple that the time spent distributing the data to the different processors exceeds the time saved on the computation). (TODO: Amdahl's law here?) Unfortunately, many important problems are not embarrassingly parallel because one computation may depend on an intermediate result from another computation (if, on the other hand, it depends on the *final* result, it cannot be said to be parallelisable). (TODO: Comm. to comp. ratio; scaleability)

In ASYNPLEX, there are four different kinds of processes:

- One *invert processor*;

- One *basis change manager*;

- One *column selection manager*;

- One or more *iteration processes*.

---

[11]In a synchronous algorithm, the code contains synchronisation points where two or more processes or threads must wait for each other to reach the point before proceeding. In asynchronous algorithms, the only kind of waiting that may occur is waiting for incoming messages from other processes or threads.

We will interleave their descriptions with the description of the general idea behind the algorithm.

Matrix inversion

The *invert processor* is continuously performing inversions of the $\mathbf{B}$ matrix. Whenever one of the iteration processes performs a pivot operation, it sends a message to the invert processor telling which variable that entered and which one that left. Once the invert processor finishes the current inverse calculation, it distributes the resulting $\mathbf{B}^{-1}$ matrix to the iteration processes. Then, it receives all incoming basis change messages and begins a new inverse calculation. Most likely, the iteration processes will find that the inverse is somewhat out of date when it is received, but they will just delete the appropriate number of entries from the eta file. This approach sacrifices some numerical stability for the increase in speed that is obtained by dedicating a separate processor to the inversion operation. See Section 3.4.1 for a small discussion of what happens if this approach is used on its own, without the other elements of ASYNPLEX.

Candidate persistence

The key observation upon which ASYNPLEX is based is a phenonomenom called *candidate persistence*. An *attractive candidate* is a nonbasic variable whose objective function coefficient is negative, so that it is possible to select it as the entering variable. According to [18], a variable that is attractive in one iteration (but remains nonbasic because some other variable is eventually selected as the entering variable) will often remain attractive in subsequent iterations. Furthermore, it can be observed that the pivot operation itself is usually very cheap (assuming that the implementation swaps matrix columns implicitly by using permutation lists to keep track of the current location of each column, while the columns themselves remain in one place) — the majority of the work in each iteration is associated with determining the entering and leaving variables and updating the solution vector. This leads to the idea of having several processes (the iteration processes) speculatively computing the $\hat{\mathbf{a}}$ corresponding to several attractive candidates. When an iteration process has completed the calculation of $\hat{\mathbf{a}}$, it sends to the basis change manager an offer to compute the leaving variable and perform the pivot operation. Given any basis, only one iteration process should be allowed to decide how to pivot away from it (otherwise, the iteration processes would diverge in different directions), and the basis change manager handles this. If the offer is accepted, the iteration process will tell all other processes which variable that left and which one that entered, and the other iteration processes will update their $\mathcal{B}$ and $\mathcal{N}$ accordingly. It also computes a new set of attractive candidates. Iteration processes that have had their

offers rejected will request new variables from the column selection manager, which keeps track of which variables are currently regarded as attractive.

(TODO: had a todo here, can't remember...)

The pseudocode uses some overly compact names (that probably stem from some old naming convention; Maros[30] uses them too) for each step of the algorithm; they are as follows (taken from [18], with some modifications):

BTRAN Compute $\pi^\top \leftarrow \mathbf{c}_\mathcal{B}^T \mathbf{B}^{-1}$ (in the process, we will use the eta file entries in reverse order, hence the name *Backwards TRANsformation*[30]).

PRICE Compute the reduced costs: $\hat{\mathbf{c}}_\mathcal{N}^\top \leftarrow \mathbf{c}_\mathcal{N}^\top - \pi^\top \mathbf{N}$.

CHUZC Choose entering variable (*Column*) by finding a negative entry in $\hat{\mathbf{c}}_\mathcal{N}$.

FTRAN Compute $\hat{\mathbf{a}} \leftarrow \mathbf{B}^{-1} \mathbf{a}_q$, where $\mathbf{a}_q$ is the column of $\mathbf{N}$ that corresponds to the entering variable (this time, the eta file will be used forwards, hence *Forwards TRANsformation*).

CHUZR Choose leaving variable (*Row*) by looking at the componentwise ratios of $\hat{\mathbf{b}}/\hat{\mathbf{a}}$, where $\hat{\mathbf{b}} \leftarrow \mathbf{B}^{-1} \mathbf{b}$. Let α be the smallest such ratio.

UPRHS Update the right-hand side by adding $\alpha \hat{\mathbf{a}}$ to $\hat{\mathbf{b}}$.

UPDATE_BASIS Add an entry to the eta file.

We now present the pseudocode for ASYNPLEX as it is given by Hall and McKinnon[18] (with a few notational adaptations), in Algorithms 3, 4, 5, and 6. It is assumed that there is a separate, sequential piece of code that handles input reading and sets up the different processes. In Section 3.4.2, we describe how we have adapted the algorithm.

A short explanation of Hall's notation may be useful. Each process has a number of points where it sends or receives data to or from the other processes. Each such communication endpoint is given a short identifying tag, both on the sending and receiving end, and each send or receive operation indicates where it wishes to send to or receive from (and the process' own tag for that operation is indicated with a comment in the right margin — note also that each type of process has its own letter). Iteration process tags are suffixed with a colon and the index of the process, since there can be several iteration processes.

2.1.5   (RP) Interior point methods

It is possible to interpret the simplex method in a geometric fashion: with n decision variables, the space of all vectors of possible decision variable values form an n-dimensional space. Each constraint can be modelled as a plane in this space — an equality constraint requires that feasible solutions lie on the plane,

loa 3: ASYNPLEX — iteration process number i $(0 \leq i < p)$

```
1:  procedure RUNITERATIONPROCESS(i, p, N, b, c)
2:      k_i ← 0
3:      BTRAN
4:      PRICE
5:      FTRAN — let q be the ith most attractive candidate column, or -1 if that
    does not exist
6:      repeat
7:          if received ← V2 an LU factorisation of the inverse then          ▷ I1
8:              Install new inverse
9:          end if
10:         while basis changes received ← I7 are not yet applied do           ▷ I2
11:             Apply basis change; k_i ← k_i + 1
12:         end while
13:         Permute column a_q
14:         FTRAN
15:         while basis changes received ← I7 are not yet applied do           ▷ I3
16:             Apply basis change
17:             FTRAN_STEP; k_i ← k_i + 1
18:         end while
19:         if q = -1 or ĉ_q > 0 then
20:             Send → C4 a message that the candidate is unattractive         ▷ I4
21:         else
22:             Send → R1 an offer to perform CHUZR                            ▷ I5
23:             Wait ← (R2 or R3) for a reply to offer                         ▷ I6
24:             if Offer accepted then
25:                 CHUZR
26:                 Send → (I2/I3/I10 on all other iteration processes) the basis
    change and pivotal column                                                  ▷ I7
27:                 Send → (V1 and C1) basis change                           ▷ I8
28:                 UPDATE_BASIS; k_i ← k_i + 1
29:                 BTRAN
30:                 Permute π
31:                 PRICE
32:                 FTRAN — choose a set of the most attractive candidates
33:                 Send → C2 the most attractive candidates                  ▷ I9
34:             else
35:                 Wait ← I7 for next basis change                           ▷ I10
36:                 goto line 15
37:             end if
38:         end if
39:         Wait ← (C3 or C5) for a new candidate column, q                   ▷ I11
40:     until The algorithm terminates
41: end procedure
```

loa 4: ASYNPLEX — invert processor

1: **procedure** RUNINVERTPROCESSOR($p, m, \mathbf{N}$)
2:     Let $\mathbf{B}$ be an $m \times m$ identity matrix
3:     $k_v \leftarrow 0$
4:     **repeat**
5:         **while** received ← **I8** a notification that x_l has left the basis and x_e has entered **do**   ▷ **V1**
6:             Swap the corresponding columns between $\mathbf{B}$ and $\mathbf{N}$
7:             $k_v \leftarrow k_v + 1$
8:         **end while**
9:         **INVERT**
10:         Send → **I1** on all p iteration processes the new LU factorisation of the inverse for basis k_v ▷ **V2**
11:     **until** the algorithm terminates
12: **end procedure**

loa 5: ASYNPLEX — column selection manager

1: **procedure** RUNCOLUMNSELECTIONMANAGER(m, n)
2:     $k_c \leftarrow 0$
3:     Mark all nonbasic variables as unselected
4:     **repeat**
5:         **if** received ← **I8** basis change **then**   ▷ **C1**
6:             Mark the variable which has left the basis as unselected
7:         **else if** received ← **I9:**i a set of candidates corresponding to basis k_i **then**   ▷ **C2**
8:             **if** $k_i > k_c$ **then**
9:                 Filter out the candidates already selected and those already rejected after the FTRAN at a basis $\geq k_i$
10:                 $k_c \leftarrow k_i$
11:             **end if**
12:             Send → **I11:**i the most attractive candidate to enter the basis and mark the candidate as selected   ▷ **C3**
13:         **else if** received ← **I4:**i a message that its current candidate is now unattractive **then**   ▷ **C4**
14:             Send → **I11:**i the most attractive candidate to enter the basis and mark the candidate as selected   ▷ **C5**
15:         **end if**
16:     **until** the algorithm terminates
17: **end procedure**

loa 6: ASYNPLEX — basis change manager

1: **procedure** RUNBASISCHANGEMANAGER
2:     $k_b \leftarrow 1$
3:     **repeat**
4:         **if** received $\leftarrow$ **I5:**i an offer to perform CHUZR for basis k_i **then**   ▷ **R1**
5:             **if** $k_i = k_b$ **then**
6:                 Send $\rightarrow$ **I6:**i an acceptance of the offer   ▷ **R2**
7:                 $k_b \leftarrow k_b + 1$
8:             **else**
9:                 Send $\rightarrow$ **I6:**i a refusal of the offer   ▷ **R3**
10:            **end if**
11:        **end if**
12:    **until** the algorithm terminates
13: **end procedure**

and an inequality constraint requires that feasible solutions lie on or to one of the sides of the plane. Together with the planes from the implicit nonnegativity constraints, this forms a (TODO: spell "geometrical") shape known as a *simplex* — hence the name of the simplex method. Each intermediate solution produced by the simplex method represents a point that is a vertex (an intersection between n or more planes). There exists another class of algorithms called *interior point methods*, whose intermediate solutions always lie in the interior of the simplex. A distinct advantage of most interior point methods over the simplex method is that they have polynomial worst-case bounds on their time consumption. The first polynomial interior point method was invented by Khachiyan[25] in 1979, and one of the most well-known methods is due to Karmarkar[23].

Interior point methods were mentioned in the problem description, but it was soon discovered that the scope of the project was already large enough even when only considering the simplex methods. Thus, interior point methods will not be taken into consideration, but we felt that no discussion of linear programming would be complete without mentioning this subject.

2.1.6   (RP) State of the art: sequential LP solvers

We now present some existing sequential solvers that we have studied.

ILOG CPLEX

CPLEX, developed by the company ILOG (http://www.ilog.com/products/cplex/), is a widely used mathematical optimisation package, and also the one currently used by Miriam AS. Being proprietary closed-source software, we cannot examine its inner workings (but they are probably too complex for this project).

GLPK

The *Gnu Linear Programming Kit* is an open source initiative to produce a versatile suite of mathematical optimisation tools. Unfortunately, the code base is extremely large, comprising more than 75000 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.

retroLP

As opposed to virtually all other LP solvers, retroLP[41] 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."[42] 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.

Numerical Recipes

The Numerical Recipes is a well-known book containing source code for the numerical solution to problems in linear algebra, differential equations and many other fields. They also provide linear algebra solvers — in the second edition[33], they use the standard simplex method, while in the third edition[34], they use the revised simplex method. For reasons to be discussed in Section 3.3, we suspect that this is due to numerical stability problems, but we found no mention of such in the book.

Vanderbei's code

Vanderbei has published a freely available implementation of the revised simplex algorithm as presented in his book[38], 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).

The code has no licence information attached to it. Anyone who wishes to commercially utilise those parts of our code that are derived from Vanderbei's code are strongly advised to contact Vanderbei.

(RP) Others

Here follows a few more solvers we are aware of. We have not had the time to study them thoroughly, but we list them here for those who might be interested in doing so.

Xpress — a commercial product, available at `http://www.dashoptimization.com/home/products/products_optimizer.html`;

OOPS — `http://www.maths.ed.ac.uk/˜gondzio/parallel/solver.html`;

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

SoPlex — An implementation developed as a part of Roland Wunderling's Ph.D. thesis[40], and available at `http://soplex.zib.de/`.

2.1.7 (RP) State of the art: parallel LP solvers

Parallel LP solvers also exist. ASYNPLEX[18] has already been discussed in greater detail, and here is a short list of some of the other solvers we have found.

Parallelisation of the revised simplex method using CUDA (Spampinato)

Compute Unified Device Architecture (CUDA) is a framework from the graphics processing unit (GPU) manufacturer nVidia. Daniele Spampinato, a student at our department, implemented the revised simplex method by using the CUBLAS linear algebra library to offload the linear algebra computations onto the GPU[37]. He reported overall speedups of 2.0–2.4 relative to a sequential implementation using ATLAS, but only for dense data sets. The only operation that (by itself) yielded the vast speedups that are theoretically possible when using GPUs (which have hundreds of cores) was the basis inversion[37, Figure 5.6 on p. 45]. Furthermore, he experienced major problems with numerical stability. Note that his implementation parallelised each linear algebra operation individually; it was not a parallel version of the simplex method itself.

SMoPlex, DoPlex (Wunderling)

These are, respectively, shared memory and distributed memory implementations of the revised simplex method, also from Wunderling's thesis[40]. Regrettably, these implementations are not available online, and since the thesis is written in German, we have not been able to study it — but it may prove useful to someone proficient in German. According to Hall[18], the implementation is "parallel for only two processors".

Parallelisation of interior point methods

Those interested in interior point methods should consult Karypis, Gupta, and Kumar[24] for an approach that allegedly scales to hundreds of processors.

retroLP

See above for a general description of retroLP, which also implements a parallel version of the standard simplex method.

Others

Again, here are some other papers and implementations we are aware of, but have not studied.

- A distributed simplex algorithm by Ho and Sundarraj[21];

- A parallelisation of CPLEX' dual simplex method by Bixby and Martin[6];

- A parallelisation of Xpress' interior point method by Andersen and Andersen[3].

2.2 Cell Broadband Engine

The *Cell Broadband Engine* (Cell/B.E.) 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[22]. Several obstacles to such goals exist; in particular the infamous *brick walls*[4]:

Memory wall While processor speeds have grown substantially over the past few decades, the growth in memory access times has been much more modest. Because of this, the relative cost of memory accesses is now prohibitively large, and for efficient scientifice computation, it is necessary to to use caches and try to keep data cached for as long as possible once it has been loaded from memory.(TODO:)

Power wall Heat dissipation becomes a greater and greater obstacle as frequency increases (TODO: and size is reduced). (TODO:)

ILP wall *Instruction-level parallelism* techniques such as pipelines and speculative execution face diminishing returns as most programs have a limited amount of exploitable parallelism, and the hardware and power cost of implementing such techniques is growing.(TODO:)

The Cell/B.E. architecture tries to solve these problems in the following ways:

- Having two different kinds of cores: one optimised for control logic and operating systems, and one optimised for computational throughput.

- Giving the programmer explicit control over data movement in the memory hierarchy, rather than having hardware-controlled caches.

- (TODO: more?)

2.2.1 Architecture

Overview

The Cell/B.E. 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 *mailboxes signals*

2.2.2 Programming methods

SIMD processing

Both the PPE and the SPE support Single Instruction Multiple Data (SIMD) operations through a data type modifier called `vector`. A `vector` is 128 bits long, and can hold e.g. four `int`s, four `float`s, or two `double`s. `vector`s must start at addresses that are divisible by 16. Declarations of automatic variables (those

located on the stack) may be suffixed with `__attribute__((aligned(16)))` to indicate such alignment. Memory allocated on the heap can be aligned by using `malloc_align()` (note that the second argument to this function is the base two logarithm of the byte boundary, so to get alignment on 16 byte boundaries, we would have to pass `4`.

There is a rich instruction set for operating on **vector**s; see (TODO: cite manual) for a full overview. All SIMD instructions are available in C and C++ as compiler intrinsics. An example is the *fused multiply-add* operation: `d = vec_madd(a, b, c)`, where all variables are **vector**s, will set each component of `d` to the componentwise sum of `c` and the componentwise product of `a` and `b`. Without the SIMD instructions, this would have had to be done with arrays and a loop: **for** `(`**int** `i = 0; i < 4; ++i){ d[i] = a[i] * b[i] + c[i]; }`.

Compiler directives

`_align_hint,` `restrict` 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`, but is translated to instructions that do not involve branches.

Double and triple buffering

Overlays

(TODO:)

2.2.3   Tools and libraries

There exist several libraries for easing the development of scientific applications on the Cell/B.E. We now provide a quick survey of the libraries that seemed to be the most relevant to us. Only one of them, the Cell Messaging Layer, was eventually used; we discuss our choice in (TODO: reference).

Accelerated Library Framework

ALF does not seem to be a good fit for our project, as

BlockLib

[1]

Cell Superscalar

(CellSs) [32]

RapidMind

http://www.rapidmind.net/

OpenMP for Cell/B.E.

[39]

MPI for Cell/B.E.

(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)

The Cell Messaging Layer (CML): http://www.ccs3.lanl.gov/~pakin/ software/cellmessaging/

(TODO: move)CML does have some disadvantages. First, it only supports messaging between the SPEs, not between an SPE and the PPE. Second, CML (like MPI) employs the Single Program Multiple Data (SPMD) model, which means that all processors must run the same program. This means that even if different SPEs are to perform different tasks, they must each contain the code both for its own functionalify and the code for the functionaligy of all other SPEs. Still, we chose to run both the column selection manager and the basis change manager on the SPEs, because their code is fairly short, their operations are fast and simple, and it is vital that they are able to respond quickly to messages from the iteration processes. For the same reasons, we merged them into one SPE thread so that the other seven SPEs would be available for iteration processes.

(TODO: Locate a BLAS library)

2.3   (RP) Miscellaneous topics

2.3.1   (RP) Data sets

`netlib`[12] is a collection of 98 LP problem sets, many of which stem from real life problems. This is our primary source for data sets to test our solvers with. They range in size from 28×32 to 16676×15695, and are mostly fairly sparse (the set with the biggest dimensions contains 74004 nonzeroes). Some of them have special properties, such as being extremely degenerate. All sets are represented in the MPS format; see Section 3.2.3 for a brief description and further references.

The official `netlib` site is `http://www.netlib.org/lp/index.html`, but it contains compressed files versions that must be decompressed with a fortran program. The collection is available in more convenient formats elsewhere, e.g. `ftp://ftp.numerical.rl.ac.uk/pub/cuter/netlib.tar.gz`.

2.3.2   (RP) Representation of sparse matrices

There are many ways of representing a sparse matrix efficiently, of which Shahnaz[36] provides a compact review. The one we ended up using in this project (see Section 3.4.2.2 for the reason why) is called the Compressed Column Storage format, also known as the Harwell-Boeing Sparse Matrix Storage Format[36]. A sparse $m \times n$-matrix containing k nonzero values is represented as two numbers telling the number of rows and columns, and three arrays:

values contains all k nonzero values, column by column (first all nonzero values from the first column, from top to bottom, then from the second column, and so on).

rowIndices contains k integers, one for each element of the *values* array, telling which row that element is located in.

columnPositions always contains $n + 1$ elements — one for each column, and one additional element. Each entry contains an index into the two other arrays, which tells where the values of the corresponding column start. The last element contains k, which in effect tells us the last valid index into the two other arrays (namely $k - 1$). Thus, the indices of the elements of column i are $rowIndices[i]$ through $rowIndices[i + 1] - 1$.

For instance, the matrix

$$\begin{bmatrix} 9 & 0 & 0 \\ 0 & 2 & 7 \\ 4 & 3 & 0 \end{bmatrix}$$

would be represented as follows:

Value	9	4	2	3	7	—
Row index	0	2	1	2	1	—
Column positions	0		2		4	5

Note that we use zero-based indices. A sparse (column) vector can be represented similarly, as two arrays containing the nonzeroes and the row indices, and two single variables telling the number of rows and the number of nonzeroes.

2.3.3   (RP) Amdahl's law

The following section is taken from our fall project report[10].

In 1967, Gene Amdahl argued[2] that parallel processing was *not* a good way to improve performance, based on the following observation: If we have a computation that consists of a sequence of serial steps that take a total time of t_s, and a certain percentage f of these steps can be performed in parallel using p processors[12], then the total time for the sequential part of the calculation will be ft_s, and if we can distribute the remaining workload equally over the p processors, the parallel part will take the time $(1 - f)t_s/p$. Since the sequential part must presumably be completed before the parallel computations can begin (or the other way around), these times must be added together for a total time of $t_p = ft_s + (1 - f)t_s/p$, and we arrive at *Amdahl's law*[13] for the speedup $S(p)$:

$$S(p) = \frac{t_s}{t_p} = \frac{t_s}{ft_s + (1 - f)t_s/p} = \frac{1}{f + (1 - f)/p} = \frac{p}{1 + f(p - 1)} \qquad (2.37)$$

The most significant aspect of this formula is that it highlights the importance of f as a severely limiting factor for the potential speedup of parallelising. Amdahl used this to claim that parallel programming was not a good idea. However, since parallelisation is currently regarded as the primary way of improving performance in high performance computing[4], the modern interpretation is that speedups can come arbitrarily close to p if only we can make f small enough, and thus, one should focus on reducing f when parallelising a calculation. On the other hand, since f in most situations cannot (even approximately) become equal to zero[14], Amdahl's law provides an upper bound on performance gain for a specific problem given the best nonzero f we can achieve: As p tends to infinity, the speedup converges to $1/f$. Again, it should be noted that these bounds are optimistic, as they are based on very simplifying assumptions.

---

[12]These are quite optimistic calculations, since we assume that the calculation can be parallelised without incurring communication penalties or extra computation steps, and that the parallel processors are as fast as the sequential one.

[13]Amdahl did not actually state this formula in his article, but it has been derived later (in many different forms) from his article.

[14]Calculations in which $f \approx 0$ and where the assumptions about independence between the parallel parts hold are called *embarrassingly parallel*.

(TODO: Actually refer to this law somewhere; fix references in chapter intro, also to pthreads)

Chapter 3

Development

(TODO: Chapter introduction)

3.1 (RP) Overall approach

The author and his advisor agreed that we should follow a step by step approach in which we begin with solving simpler problems and gradually proceed towards more challenging problems. We decided upon the following overall plan:

1. Implement the standard simplex method on a sequential machine, in order to gain familiarity with the subject and produce a program on which we can base our later development.

2. Parallelise the standard simplex method on Cell/B.E. (if the Cell/B.E. turns out to be very hard to program, we could first parallelise it on a regular multicore machine using e.g. pthreads or OpenMP (see `http://openmp.org/wp/`) to make sure our parallelisation approach is correct).

3. Implement the revised simplex method on a regular (single core) machine.

4. Parallelise the revised simplex method on Cell/B.E.. This is expected to be much harder than the parallelisation of the standard simplex method, and we may have to resort to implementing one or more preexisting parallel simplex methods. For each implementation, several possibilities for refinements could be investigated:

   a) Investigating various ways of handling numerical instability that may occur when using single precision arithmetic.

   b) Experimenting with how different representations of sparse matrices and vectors affect performance.

   c) Figuring out how to improve performance by utilising the Cell/B.E.'s
      SIMD operations.

   d) Using autotuning to find good values for e.g. data transfer block
      sizes.

5. Investigate interior point methods and implement them if time permits.

3.2   (RP) Standard simplex method

In order to become familiar with programming the Cell/B.E., we initially imple-
mented a few versions of the standard simplex method (which is best suited for
dense problems). We began with a straightforward implementation for regular
(sequential) x86 machines, then ported it to run on the Cell/B.E. PPE while util-
ising its SIMD instructions. Finally, we created a parallel version of the method
which uses the SPEs. However, as will be discussed in Section 3.3, it turned out
that it is extremely hard to make the standard simplex method work reliably on
even medium-sized data sets.

(RP) Choice of development language   The Cell/B.E. SDK offers three lan-
guages: assembly, C, and C++. Assembly language is of course out of the
question (due to development time and risk), except perhaps for a few very
performance-critical parts — but modern compilers are normally very good at
optimising, and there are C and C++ intrinsics for the extended instruction sets
of the Cell/B.E.. Although we do not need advanced object-oriented features
that cannot easily be emulated in C, C++ is the language of our choice, for the
following reasons:

   • The author is more familiar with C++ than with C.

   • It offers more high-level data structures than C, through the Standard Tem-
     plate Library — in particular, the `vector` class for dynamic arrays.

   • The author prefers the syntactic sugar of working with classes with mem-
     ber methods rather than functions that take structures as parameters.

   (TODO: Our Cell/B.E. implementation is a fairly straightforward paralleli-
sation of the standard simplex method (that is, it is only a parallel formulation of
the same algorithm), so it gives the same results as our sequential implementa-
tion. Given that the sequential implementation normally yields a wrong answer
for problems of nontrivial size, it is not particularly useful for computations, but
Miriam (who has invested in a Cell/B.E. (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 4.2.1 we provide detailed timings of some test runs.)

3.2.1 (RP) x86 and PPE version

Our x86 solver is a more or less direct implementation of the pseudocode from Vanderbei[38] and Cormen et al.[7]. However, there are some refinements:

- Rather than a raw array, we use a class for representing the tableau, so that we can make functions for operations such as adding a multiple of a pivot row to another.

- We would like the ability to switch between different data types in order to test how much the numerical stability is affected by the use of single precision arithmetic rather than double precision. Therefore, we use the symbol `TYPE` as a data type. This symbol is expected to be defined as `float` or `double` or any other suitable data type (see Section 3.3.2) — this can be done through a compiler switch, e.g. `-DTYPE=float`.

- During development, all functions that accessed the tableau performed bounds checking, in order to facilitate debugging. This incurs a performance penalty, in particular in simple functions that just read or write one entry, so we removed the code after the development was complete. In hindsight, we should have used defines to allow for simple enabling or disabling of that feature, but it will not be hard to add such code again.

The PPE solver is very similar. 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 might be the low arithmetic intensity of the simplex method, 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. However, the Cell/B.E. has an advantage in this situation over other architectures, since the LS is faster than regular caches. On the other hand, once the tableau becomes too large to fit in the LS, one may start seeing reduced payoffs from SIMD instructions because data cannot be provided fast enough.

We only need one type of SIMD instruction, namely the fused multiply-add, as described in Section 2.2.2. The central part of the code is the following line: `destination_v[j] = vec_madd(source_v[j], factor_v, destination_v[j]);`, which multiplies four values from the source row by a specified factor and adds them to the destination row.

An interesting implementation detail related to SIMD is that **vector** instructions can only operate on data that is aligned on 16-byte boundaries (that is, four. Therefore, we must use `malloc_align()` for memory allocation. Furthermore, one must keep in mind that the width of the tableau may not be divisible by four, in which case the last elements will not fill an entire **vector**. The simplest way of getting around this (and avoiding tests for whether one has reached the end of a row) is to round the number of columns up to the nearest multiple of four and allocate the corresponding amount of memory. To the other parts of the code, the width will still appear to be the original number of columns, whereas all internal operations will utilise the fact that each row contains an integral number of **vector**s. The last elements in a row may be garbage, but that does not matter since they are not reachable from the rest of the code (and there are no division operations, so division by zero is not a risk). This is implemented in the `Matrix` class (see Appendix A.1) by having one variable called `cols`, which keeps track of the original number of tableau columns, and one called `physicalCols`, which keeps track of the rounded-up width.

(RP) Code   The PPE code was developed by copying and rewriting the x86 code, but we were able to merge them back into one program (through the use of **#define**s that control which parts of the code are activated), listed in Appendix A.1. This sacrifices a little bit of readability for the sake of compactness and avoidance of code duplication. The appendix also contains information on how to compile it, and short descriptions of what each file contains.

3.2.2   (RP) SPE version

Our approach is fairly obvious[1]:

1. The PPE, which initially holds the entire tableau, distributes the tableau rows evenly among the SPEs, 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.

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

---

[1]After having written the application, we found that Yarmish[41] uses a very similar approach, albeit for cluster computers with MPI.

4. The PPE determines which SPE that "wins" (because it has the strictest bound) and requests this SPE to transfer the pivot row to main memory using DMA; afterwards, all the other SPEs are asked to initiate a DMA transfer to receive this row. 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 notifies the PPE upon completion. Go to step 2.

We would have liked to employ some sort of broadcast operation for distributing the pivot row in step 4, but we could not find out how to do so.

Note that we have only implemented Phase II. The code is listed in Appendix A.2. SIMD operations are utilised on the SPEs, in a manner similar to what was done in the sequential PPE solver.

3.2.3 (RP) MPS and CPLEX parsers

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, we had to write our own. We also needed to write a parser for the ILOG CPLEX format, since Miriam AS provided us with a few data sets in that format (see Appendix B.2). Our parsers do not handle all aspects of the formats, but they are sufficient for those sets that we use. The source can be found in Appendices A.6 and A.6. Maros[30, Chapter 6] gives a fairly compact presentation of the MPS format.

3.3 Implementation problems

(TODO: Put the schedules somewhere (original, revised, and after meeting?))

3.3.1 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 the lines described in Section 3.1. 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 [7] and [38]), and the Cell/B.E. 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 [7] and [38] make no mention of the standard simplex method being particularly *unstable* (they only say that other methods are being used in practice because they are more *efficient*). Agreeing with the words of Donald Knuth, "premature

optimization is the root of all evil"[27], we decided that (TODO:) Time spent optimising an incorrect program is most likely going to be wasted time, as the optimised parts will probably be rewritten (or the whole program is discarded and another one is developed from scratch) , and

Also, [33] 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[34], 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[42]. However, the code base was quite large, and (TODO:)

We eventually resigned and contacted a group of mathematicians with which Lasse is acquainted, describing our problems and asking them for help on how to make the standard simplex method work stably[19]. Their response (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[30]. (TODO:)

3.3.2   Numerical stability

(TODO: something on float vs. double?) Already at the beginning of the project, concerns were raised about the suitability of the Cell/B.E. for this kind of computations, since its double precision arithmetic is very slow compared to its single precision arithmetic.

Approaches we have tried include the following:

- Scanning the tableau in-between each iteration and setting every number whose absolute value is below a certain threshold to zero.

-

- Terminating Phase I when x_0 has reached a value that is "sufficiently close" to zero.

- Whenever two numbers are added to or subtracted from each other, the result is compared to the two original numbers. If the ratios between the result and each of the original numbers are sufficiently small, the tresult is set to zero. We had some success with this rule; if the ratio is e.g. 0.00001, the right result for AFIRO is produced.

For some data sets, we were able to find values where None of these yielded consistently improved results

In order to demonstrate that the stability problems are not caused by errors in our implementation, we have made our code support usage of the *GNU multiple precision arithmetic library* (GMP — see `http://gmplib.org/`), which among other features 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[2]. Compile the code by running the `buildgmp.sh` script; this will link to GMP (which must first have been downloaded, compiled and installed on the system) and tell our code to use the `mpq_class` data type for all arithmetic operations and to output results in fraction form. When using GMP, the code obviously slows down by a significant factor and the memory consumption increases (which is why this approach is useless in practice unless it is absolutely essential to obtain exact results). Table 3.3.2 on the next page shows the results for some of the small-to-medium `netlib` sets. Note that our solver performs maximisation, while the `netlib` sets are supposed to be minimised (but for some reason, the MPS format does not specify whether to maximise or minimise) — therefore, our MPS parser negates the objective function, so that the answers will have correct absolute value but wrong sign. According to the `README` file, The "official" `netlib` results have been obtained using the MINOS solver, version 5.3. All digits of the `netlib` results agree with our exact results.

3.3.3 (TODO: Missing citations)

(Natvig/Elster/Mujahed: This section is quite randomly placed; it is simply a list of articles I think I am going to cite, but where I haven't yet written the context in which they are to be cited.) [17] [13] [16] [28]

3.4 Revised simplex method

(TODO:)

3.4.1 (RP) 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

---

[2]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.

Data set	Netlib result	Our result	Iterations	Time
AFIRO	$-4.6475314286 \cdot 10^2$	$\frac{406659}{875}$	16	0.044 s
BRANDY	$1.5185098965 \cdot 10^3$	$\frac{-16065877392598163704545292298\,35255763845946280057831648209\,5777480900411096633986368891}{1058002811160721713504750150\,8720411569323127506371426417\,345909327662918125000000000}$	605	491.402 s
LOTFI	$-2.5264706062 \cdot 10^1$	$\frac{631617651547}{25000000000}$	537	40.362 s
SCFXM2	$3.6660261565 \cdot 10^4$	$\frac{-4874671419119861011078305839244\,65\,3390630042031652016001773580110200\,0732423011933261045459132101058706\,9407177301915047835480055104995559}{1329688117603047126754336400784\,88\,8771958942099169754747473929704\,67\,4848158506258498441472830720462\,61\,3814446552258600000000000000000\,0000}$	1299	2363.2 s
STOCFOR1	$-4.1131976219 \cdot 10^4$	$\frac{7368963026860358678147\,0598121420626868798940\,69612494322055836783}{1791541205690536804\,8\,97461796875000000000\,00000000000000000000}$	135	3.381 s

Table 3.1: Some results of our exact standard simplex implementation

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[21, Table 2], the inversion consumes less than 20% of the total time of the revised simplex method, and as such, speedups are limited as per Amdahl's law (see Section 2.3.3). 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.

3.4.2   Our adaptation of ASYNPLEX

As discussed in (TODO: reference), we realised that we had too little experience with numerical computation in general and the simplex method in particular, and therefore we decided to find an existing sequential implementation of the simplex method and rewrite it as per the ASYNPLEX algorithm. Finding a suitable implementation was not easy, because one apparantly needs to make a trade-off between small code size and ease of understanding on one hand and numerical stability on the other hand. As noted in (TODO: reference), all the major, well-known implementations have exceedingly large code bases.

(TODO: Have we discussed rationales for our choice anywhere?) After unsuccessful attempts at understanding GLPK and (TODO:), we chose Vanderbei's implementation.

Basing ourselves on an existing sequential implementation also has the advantage of allowing a direct comparison between the sequential and parallel versions of the same code, allowing us to better (TODO: spell "gaugue") the speedup that is offered by ASYNPLEX itself (Hall et al. measured their performance against an entirely different sequential implementation), and the further speedup that is obtained on Cell/B.E.. (TODO: actually perform such comparisons)

The disadvantage, of course, is that retrofitting an sequential implementation may require a vast effort in case parts of the code does not lend itself well to parallelisation (this easily happens when the code relies on global variables or internal, static variables, because such variables will need to be duplicated so that each thread has its own instance of it.) For this reason, we chose to switch to C++(TODO:).

3.4.2.1 ASYNPLEX prototype in C#

Appendix A.3

3.4.2.2 (RP) Sparse vector and matrix representations

Vanderbei's implementation uses the Compressed Column Storage format (as described in Section 2.3.2) for sparse matrices and a similar scheme for sparse vectors. Unfortunately, he did not have a structure or class that contained the arrays and variables for each sparse matrix or vector. For instance, the matrix **A** would be represented with the arrays `a` (values), `ia` (row indices), `ka` (column positions) and the variable `na` (TODO:) (number of nonzeroes) — a naming scheme that we found to be very impractical (all variables must be passed as parameters to functions that are to manipulate sparse vectors and matrices), and which (TODO: oppos. facilitate) our process of understanding his code. Therefore, we introduced structures that combined these related arrays and variables, and we refactored the code to use these strucures throughout. Our structure for sparse matrices looks like this:(TODO: remove typedefs)

```
struct SparseMatrix {
  int rows;
  int cols;
  int numNonzeroes;
  int * rowIndices;
  int * colPos;
  TYPE * values;
};
```

Note that `TYPE` is a preprocessor symbol which facilitates experimentation with different precisions (TODO: describe TYPE somewhere) — it should be defined as either **`float`** or **`double`**.

Due to the vast amounts of vector manipulation (and also in order to track down some bugs we believed were related to reading/writing outside of the array bounds, but turned out to be caused by wrong memory management), we made a more elaborate sparse vector structure, which uses the **`vector`** class from the C++ Standard Template Library. The `at()` function performs boundary access checking on each access. (TODO: a define to enable/disable usage of at(), and update this code) The compiler will most likely inline the simple accessor functions and operators, so that the usage of classes with and **`vector`** will not incur any performance penalty (if the boundary checking is turned off). The structure can be found in (TODO: file name, appendix ref).

Beware that in order to save time, Vanderbei preallocates the arrays for any sparse vector with r rows to have size r, but only the first k entries are used at any time (where k is the number of nonzeroes). Whenever the contents (and the number of nonzeroes) of the vector changes, one can simply fill the arrays with as many entries as necessary, since each individual vector has a constant size throughout the program and the number of nonzeroes obviously will never exceed the full vector size. Also, Vanderbei did not explicitly store the sizes of the vectors and matrices, as they could always be deduced from context (normally as having m or n rows). We feel that this practice obscures the relationship between a loop header and its body — if `v` is a sparse matrix with `n` columns and we want to write a loop that manipulates `v`, we prefer e.g. **`for`** (**`int`** `j = 0;` `j < v.cols; ++j`) to **`for`** (**`int`** `j = 0; j < n; ++j`). Therefore, we have included the size information into our structures and have tried to used them instead of `m` and `n` (this also makes the linear algebra functions slightly more general, and it would facilitate unit testing). Note that such preallocation is not done for matrices, since this would require too much space, and because the main part of the algorithm never changes the matrices directly (it uses permutation lists to keep track of how columns are swapped).

3.4.2.3   Restructuring of Vanderbei's code

3.4.2.4   Communication approach

ASYNPLEX is an algorithm for message-passing distributed memory systems, but its authors describe how to adapt it to shared memory systems. While the Cell/B.E. architecture resembles both shared memory and distributed memory architectures, we chose to go with the message-passing approach

3.5 Debugging and testing

3.5.1 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[3] 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 .

Any commercial product ought to have thorough unit tests, but While some of these considerations are most relevant for software companies, (TODO:)

3.5.2 Array bounds checking

`mudflap`

3.5.3 Memory leaks

`valgrind` with `MemCheck` (`http://valgrind.org`) is an invaluble tool for detecting memory leaks (forgetting to release memory segments that are no longer in use, such that the program will continuously consume more and more memory) and illegal use of the memory allocation system (such as calling `free()` on the same pointer twice, which may easily cause corruption of the memory allocator's internal data structures). We have used this tool on several occasions during this project, and we believe that we have removed all memory leaks caused by our own code. The only leaks that remain are caused by Vanderbei's own code, but each leak occurs only once (not inside loops) and the data that is being allocated is needed throughout the entire program (and is automatically

---

[3]As with many other good practices, unit testing is easier to preach than to practice, as evidenced by the lack of unit testing in this project.

freed when the program terminates), so that we chose not to spend time on re-moving them. A `valgrind` report on one run of our (TODO: x86) solver can be found in Appendix (TODO:).

3.5.4 Comparison to other implementations

Miriam currently uses the ILOG CPLEX solver, and it would therefore be rea-sonable 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[18] provides relative speedups of ASYNPLEX runs on a few `netlib` sets, with which we can compare our speedups.

Chapter 4

Evaluation

Due to all the challenges we have faced, we have not been able to produce sufficiently stable solvers, and our Cell/B.E. implementations only handle very small data sets. Still, a number of interesting questions can be posed, and their answers might serve as a guidance to those that will continue the project.

4.1 Performance measurements

4.1.1 (RP) Testing environments

The x86 experiments were run on a machine containing an Intel Core2 Quad Q9550 with four cores at 2.83 GHz, with 4 GB of system memory. The compiler is `gcc` version 4.2.4. The system is running Ubuntu version 9.04 "jaunty" with Linux kernel version 2.6.28-11-generic.

The Cell/B.E. experiments were run using the IBM Full-System Simulator, version 3.1-8.f9, on a computer running Fedora 9. Being a simulator, the timing results obtained on it are independent of the physical hardware of the host computer. The simulated PPE cores have a frequency of 3.2 GHz.

4.1.2 What to measure (TODO: reword — "research questions?")

(TODO: Is all of this covered below?)

- How the numerical stability and the accuracy of the answer is affected by using single precision in place of double precision

- Speedup of the Cell/B.E. parallel standard simplex implementation relative to the x86 implementation

- How the speedup of the Cell/B.E. parallel standard simplex implementation depends on the number of SPEs used (relative to a version using only the PPE)

- How well a vectorised Cell/B.E. parallel standard simplex implementation the performs relative to a non-vectorised version

- Speedup of the x86 ASYNPLEX implementation (on a multicore) relative to Vanderbei's original solver

- Speedup of the Cell/B.E. ASYNPLEX implementation relative to the x86 implementation

- Time spent waiting for data to be moved to the local store (both for the standard simplex method and for ASYNPLEX)

We will use data sets ranging from the smallest ones to the largest that our solver can handle, in order to test the scalability of our implementations. (TODO:)

4.1.3   (RP) Measurement methods

All Linux distributions incorporate the `time` utility, which reports the amount of time spent by a process: `real` (wall time), `user` (time spent in the process' own code), and `sys` (time spent in system calls on behalf of the process). For multi-threaded programs, the two latter values will be the sums of the time spent by all threads and may therefore exceed the first value. The precision is at most one millisecond, and it is limited to timing an entire program. In order to time only parts of a program, the standard C++ function `clock()` is commonly used. Unfortunately, its resolution is system-dependent and often too coarse. Therefore, (TODO:) we opted for a much more high-precision timer that is available on Intel processors and PowerPC (and thereby on the PPE), called the Time Stamp Counter. This counter is incremented on each clock cycle (on PowerPC, it might be controlled by a separate clock[44]), and it may be read by using the `rdtsc` instruction. Timing utilities have been implemented in `timer.h`, and they employ the function `rdtsc()`, taken from [44], that uses the aforementioned instruction. The drawback is that in order to get the time in seconds, we must emprically determine the `rdtsc` frequency. Using the following simple code fragment (which was run with `time` in order to verify that it actually slept for that long; we also tried several different delays), it was found to be 25 MHz on the PPE (we later discovered that this could also be found with the command `cat /proc/cpuinfo`) and 2.83 GHz (as expected) on the machine described in Section 4.1.1:

```
#include <unistd.h>
#include <cstdio>
#include "timer.h" // See the source listings in the appendix
int main() {
  unsigned long long start = rdtsc();
  usleep(1000000); // Sleep for one second
```

```
  printf("%llu\n", rdtsc() - start);
  return 0;
}
```

All x86 and PPE programs were compiled with the `-O3` switch (maximal optimisation level).

The Cell/B.E. simulator can gather detailed statistics on each of the SPEs. Before invoking the program one wishes to analyse, all SPEs must be set to "pipeline mode", and after the program has been run and the simulator has been stopped, the simulator command (TODO: mysim spu X stats print) can be issued for each SPE by replacing the X by a number between 0 and 7 to indicate which SPE one wants the statistics for.

4.2 Results

Note that the timings do not include the reading and parsing of the MPS input file, as this must necessarily be done by any implementation, and is not of interest when one desires to gauge the efficiency of the core algorithm. We do, however, include the time required for starting and stopping the SPE threads, as this is a Cell/B.E. specific feature, and we felt that excluding it would give the Cell/B.E. an unfair apparent advantage in comparison to other architectures.

4.2.1 Standard simplex method

As discussed in (TODO: reference), the standard simplex method is highly susceptible to numerical instability, and our implementation is no exception to this. It is essentially useless in practice because for most sets of realistic size, it produces answers that are off by orders of magnitude. Still, we might be able to learn something about the computation to communication ratio of the algorithm, and how much time vector operations are capable of saving. Also, Miriam stated that they are interested in such measurements.

It should be noted that these timings were obtained while running the simulator in "fast" mode, rather than in "cycle" mode (which is what one should ideally use for benchmarking), because we had problems getting the latter mode to work[1]. These modes control how closely the simulator mimics the real hardware, and it could be that the results will be different if the experiments are repeated in "cycle" mode.

Again, please take note that our solvers did not produce the right answer on most of the sets used here (and some sets may only have been solved to the end of Phase I, because the solver never finds a feasible solution because of the nu-

---

[1]In general, we have struggled a lot with the Cell SDK and simulator, whose installation procedures, user interfaces, and documentation are often somewhat obscure and lacking in detail.

merical instability). These analyses are merely for evaluating the performance benefits of our design and implementation approaches, without regard for numerical stability — in the hope that more stable implementations may benefit from our observations.

4.2.1.1   Numerical stability

Numerical stability is not something that is easily quantified. (TODO:)

4.2.1.2   (RP) Speedup of sequential PPE version by using SIMD

The standard simplex method has very low *arithmetic intensity* (number of arithmetic instructions per load from memory) — so much time is spent moving data from main memory into the cache and from there into the registers that the SIMD speedup of the simple operation that is executed on the data once it is in the registers may not have much impact. In order to find out how much impact the SIMD operations have in this situation, we compared the run times of the sequential standard simplex method on the PPE with and without SIMD operations. The results are seen in Figure 4.1 on the facing page. As expected, they are far away from the fourfold speedup that should in theory be possible. Note, however, that the speedup increases with the size of the data set. We did not test larger sets than these because the simulator is terribly slow, but we suspect that once the data set is too large to fit in the cache, additional slowdowns will be incurred (but this, of course, will apply whether or not SIMD is used; it will just further diminish the gains from SIMD).

4.2.1.3   (RP) Time consumption depending on number of SPEs

For our parallel standard simplex solver, it will be interesting to see how the number of SPEs (which can be anything between 1 and 8) affect the run time. Since that solver only implements Phase II, we must limit ourselves to `netlib` sets whose right hand sides are nonnegative, such as the `SC` sets. The SPEs do not currently track the locations of each variable, and therefore they cannot employ Bland's rule. Because of this, the solver cycles on `SC205`, so we have omitted that set.

The results, which are averaged over five runs, are presented in Figure 4.2 on page 54, and they are not very encouraging: going from 1 to 2 SPEs gives a minor speedup (but only for the largest set), but increasing the number of SPEs beyond 2 actually causes a *slowdown*. By looking at the detailed SPE timing statistics (as described in Section 4.1.3), in particular the overview of what the SPE spends its cycles on, we get the explanation (note that these are SPE cycles and are not supposed to match the cycle counts in the graph):

```
Single cycle                              507683 (  4.9%)
```
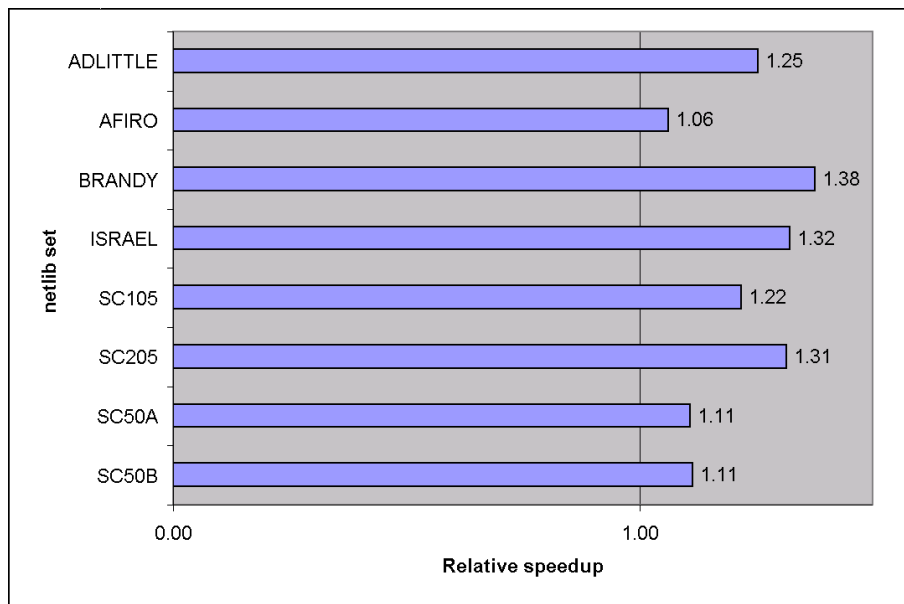
Figure 4.1: Speedup obtained in the sequential PPE standard simplex solver by using SIMD, relative to not using SIMD. Higher is better; 1.0 means no speedup.

```
Dual cycle                              89774 (  0.9%)
Nop cycle                               62087 (  0.6%)
Stall due to branch miss               183518 (  1.8%)
Stall due to prefetch miss                  0 (  0.0%)
Stall due to dependency                733134 (  7.1%)
Stall due to fp resource conflict        3024 (  0.0%)
Stall due to waiting for hint target     1324 (  0.0%)
Issue stalls due to pipe hazards            6 (  0.0%)
Channel stall cycle                   8812670 ( 84.8%)
SPU Initialization cycle                    9 (  0.0%)
----------------------------------------------------
Total cycle                          10393229 (100.0%)
```

The SPE spends the majority of its time waiting for data, and only 5.8% actually performing computations! This highlights the importance of overlapping communication and communication (which we have essentially ignored in the development since there is not much use in optimising a program that does not produce correct answers). Furthermore, this algorithm will most likely perform better for sets that have much more rows than columns, since the pivot row to be transferred will be narrower. Most `netlib` sets are either approximately square or have more columns than rows, but such sets could be handled by using the dual simplex method instead (which in effect solves the transpose of the original problem). Also, if it is possible to perform some sort of DMA
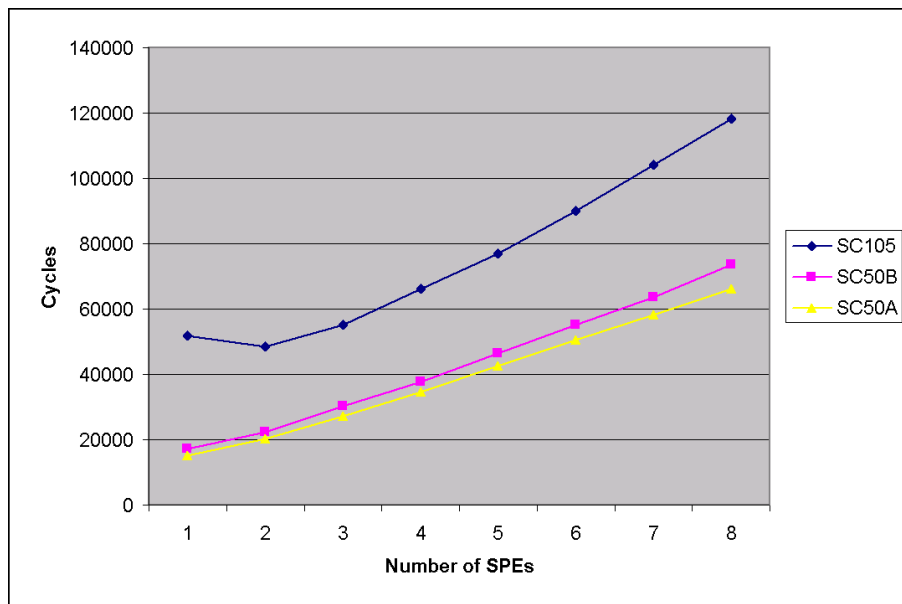
Figure 4.2: Time consumption (in PPE timer cycles) of parallel SPE standard simplex solver, depending on the number of SPE threads used. Lower is better.

broadcasting operation so that the pivot row could be distributed to all SPEs simultaneously, performance could be improved (but it could be that the MFC is advanced enough to be able to notice that all SPEs ask for the same memory region and then performs an efficient broadcast).

4.2.1.4 Speedup of parallel SPE version vs. sequential PPE version

4.2.2 ASYNPLEX

4.2.2.1 Speedup of x86 ASYNPLEX vs. original Vanderbei solver

4.2.2.2 Speedup of Cell/B.E. ASYNPLEX vs. x86 ASYNPLEX

4.3 Other aspects

4.3.1 Code size

(TODO: Not entirely sure about this) Vanderbei's Phase I/II scheme causes the code size to be larger than necessary, because much code must be duplicated and changed slightly in order to work with both $\mathbf{A}$ and $\mathbf{A}^\top$.

It is necessary to The debug flags, in particular `-g3`, dramatically increase code size. Optimisation flags, in particular `-O3`, reduce code size greatly (strangely, `-Os` does not seem to have any effect). As often happens in optimisation, there are tradeoffs that must be considered — for instance, while loop unrolling gives

a good speedup of tight loops, it increases code size, which one cannot always afford on Cell/B.E.. Thus, neither manual unrolling nor `-funroll-loops` should be done if (like us) one has a large program.

When optimising for size, one would normally want to consider using `-fno-inline` in order to disable function inlining (replacing calls to short functions by the actual function code). However, this flag actually increased the size of the object files. We suspect that this is due to heavy use of `std::vector` and its `[]` operator — the operator code itself can probably be translated into one instruction (load using memory address and offset), while a function call would require several instructions for parameter passing, stack management, etc.

The option that by far had the greatest impact on the final code size was `-s`, which tells the linker not to include symbol information (a debug (TODO: and linking) aid) in the object files. (TODO: This reduced the spu program to one-third of its original size.)

4.3.1.1 Overlays

By using overlays wisely, it may be possible to significantly reduce the size of the (TODO: I think they have a word for this - active or resident set?) that at any time resides in the local store, and thereby For instance, because of the SPMD approach mandated by CML, the code for all four ASYNPLEX processes must reside in the same program. With overlays, each SPE will only need to load the code for the process type that it is responsible for. Furthermore, since the code for each phase i slightly different, but the phase change only occurs once, it should be beneficial to only load the code for one phase at a time. We doubt that even more fine-grained use of overlays will help, unless the linear program is extremely large, in which case it may also be possible to use overlays for the code of the individual linear algebra operations that are executed in each iteration.

4.4 (RP) Reflections on unimplemented features. Ideas for future work

We will now discuss features we believe will be useful, but that we did not get the time to implement.

4.4.1 (RP) Interior point methods

The most time consuming step of many interior point methods is a Cholesky factorisation. Monien and Schulze[31] discuss approaches to parallelising this operation for sparse matrices, and one of those methods, called the *multifrontal method*, is elaborated by Schulze[35].

Andersen and Andersen[3] present a parallel shared memory version of the interior point method that is (or was at the time) underlying the Xpress solver (see Section 2.1.6). Yet another parallel interior point method is presented by Karypis et al.[24].

The opportunities for implementing any of these methods on Cell/B.E. should be investigated, but, given our experiences, it will probably be necessary to dedicate at least an entire master's thesis to that subject.

For dense matrices, Cholesky factorisation has already been implemented by (TODO: citation), whose work should be studied by those who wish to implement the same operation for sparse matrices.

4.4.2   (RP) Mixed precision

It may be possible to overcome the limited precision that is offered by the Cell/B.E. without rewriting the computation to use double precision (which will incur a massive slowdown). Some linear algebra problems can be solved by using a technique called *iterative refinement*. Kurzak and Dongarra[29] describe a successful implementation of a Cell/B.E. program for solving equations of the form $\mathbf{Ax} = \mathbf{b}$, which meets the LINPACK benchmark's requirements for the precision of the solution. Similar techniques may be investigated for the linear algebra operations that underlie the revised simplex methods.

4.4.3   (RP) Parallel linear algebra operations

It is be possible to utilise parallel BLAS libraries to parallelise the linear algebra operations themselves. Such an approach was tried on the CUDA architecture by Spampinato[37] (see Section 2.1.7), but the speedups were limited. Furthermore, as long as the number of threads is limited (such as it is on Cell/B.E.), it may be difficult to combine with an algorithm that itself is parallel (and therefore requires several threads of its own), such as ASYNPLEX.

4.4.4   (RP) Loop unrolling

Loop unrolling consists of duplicating the body of a loop such that several iterations are performed sequentially inside the loop body. This reduces the number of jumps and may yield good speedups in short, tight loops. Manual loop unrolling is usually not necessary, as compilers can be instructed to perform this operation automatically, but they may not always succeed. Unfortunately, the price that necessarily must be paid when loops are unrolled is an increase in code size, which is undesirable on Cell/B.E..

4.4.5 (RP) Representation of sparse matrices

Sparse matrices and vectors can be represented in numerous ways; Shahnaz et al.[36] 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. The first alternative representation to try might be the *jagged diagonal storage*, which, according to [36], is "specially tailored for sparse matrix-vector multiplications", and its variation *transposed jagged diagonal storage*, which is "suitable for parallel and distributed processing".

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 described in Section 2.3.2.

4.4.6 (RP) Vectorisation

As mentioned in Section 2.2.2, utilising vector operations is essential in order to obtain the high computational throughput that is promised by the Cell/B.E.. While vectorisation of dense matrix-vector operations is fairly trivial (as seen in our parallel standard simplex solver), putting vectors to good use in sparse operations is much harder. For instance, vectorisation of a simple addition of (mathematical) vectors will require the opportunity to add four adjacent numbers to four other adjacent numbers simultaneously, but with sparse representations, adjacent numbers in one vector may not correspond to adjacent numbers (or any numbers at all) in the other vector.

One approach may be to, for each nonzero number, store all four numbers that are located in the same `vector` (even if the other three are zeroes) — if the element at index i is nonzero, we would store all elements from $\lfloor \frac{n}{4} \rfloor \cdot 4$ through $\lfloor \frac{n}{4} \rfloor \cdot 4 + 3$. This would permit operations on four adjacent numbers — but only if there is a matching `vector` in the other vector. Thus, the gains from this approach may be rather limited. Furthermore, it would come at the cost of an increase in the storage requirements, which may be detrimental since it would increase the traffic on the Cell/B.E. bus. When using the compressed column storage format as described in Section 2.3.2, the required space would increase

from $2k + 1$ elements to $5k + 1$ elements (it is sufficient to store the row index of each `vector`, so only the value array would quadruple its size) in the worst case of a vector having k nonzeroes with none of them spaced closer than four elements apart. For an $m \times n$-matrix containing k nonzeroes it would increase from $2k + n + 3$ elements to $5k + n + 3$ elements.

4.4.7   (RP) Autotuning

Autotuning is the process of experimentally finding good values for compile-time constants such as block sizes for data transfer and matrix multiplication[2]. As implied by the word "auto", the experimentation is performed by another program, which repeatedly recompiles the target program with new parameters and runs benchmarks; the best parameter combination is then used for the final compilation.

A well-known software product that utilises autotuning is ATLAS, which is a BLAS library which can be automatically optimised for any architecture.

However, the benefits from autotuning may be smaller on Cell/B.E. than on regular computers, since one of the points of the Cell/B.E. architecture is that it should be possible to produce code that is predictably good because it utilises the manually-controlled memory hierarchy well — but this, of course, is complicated, and using autotuning may be simpler in some situations.

(TODO: We feel that Kurzak and Dongarra expressed this very eloquently: "Great effort has been invested throughout the years in optimizing code performance for cache-based systems, in most cases leading to the programmers reverse engineering the memory hierarchy. By requiring explicit data motion, the memory design of the Cell takes the guesswork out of the equation and delivers predictable performance."[29])

4.5   Discussion

---

[2]A typical matrix multiplication is to divide each matrix into blocks and perform the multi-plication blockwise, with the goal of having each block stay in the cache for as long as possible.

Chapter 5

Conclusion

The purpose of this project was to explore how linear programming algorithms, primarily variations of the simplex method, might be parallelised and implemented on the Cell Broadband Engine, a multicore processor with an innovative architecture. To the surprise of both the author and his advisor, the various simplex method turned out to be exceedingly difficult to implement sequentially, even on a regular computer and without parallelisation — a fact which we later learned is well-known within the mathematical optimisation community. (TODO:)

Thus, the project was turned into an exercise in reading and refactoring other people's code (a useful skill to have, but it does not exactly qualify as research)

The author selected this project (from a collection of many project proposals provided by his institute) in the belief that the algorithms , and that he would spend most of his time

5.1 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.

5.2 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. 33]

[2]   G. AMDAHL, *Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, in Proceedings of the AFIPS spring joint computer conference, 1967, pp. 483–485. [cited at p. 35]

[3]   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. 30, 56]

[4]   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. 30, 35]

[5]   H. Y. BENSON AND D. F. SHANNO, *An exact primal-dual penalty method approach to warmstarting interior-point methods for linear programming*, Computational Optimization and Applications, 38 (2007), pp. 371–399. [cited at p. 17]

[6]   R. E. BIXBY AND A. MARTIN, *Parallelizing the Dual Simplex Method*, INFORMS Journal on Computing, 12 (2000), pp. 45–56. [cited at p. 30]

[7]   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, 16, 39, 41]

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

[9]   D. P. DOBKIN, R. J. LIPTON, AND S. P. REISS, *Linear programming is log-space hard for P*, Information Processing Letters, 8 (1979), pp. 96–97. [cited at p. 16]

[10] Å. ELDHUSET, *Edge detection on GPUs using CUDA.* Fall project report, Norwegian University of Science and Technology, January 2009. [cited at p. 35]

[11] 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]

[12] D. GAY, *Electronic mail distribution of linear programming test problems*, COAL Newsletter, (1985), pp. 10–12. [cited at p. 34]

[13] P. E. GILL, W. MURRAY, M. A. SAUNDERS, AND M. H. WRIGHT, *A practical anti-cycling procedure for linearly constrained optimization*, Mathematical Programming, 45 (1989), pp. 437–474. [cited at p. 43]

[14] J. GONDZIO AND A. GROTHEY, *A New Unblocking Technique to Warmstart Interior Point Methods based on Sensitivity Analysis*, SIAM Journal on Optimization, 19 (2008), pp. 1184–1210. [cited at p. 17]

[15] 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. 16, 17]

[16] J. A. J. HALL, *Towards a practical parallelisation of the simplex method*, Optimization Online, (2005). [cited at p. 43]

[17] J. A. J. HALL AND K. I. M. MCKINNON, *PARSMI, a Parallel Revised Simplex Algorithm Incorporating Minor Iterations and Devex Pricing*, in PARA '96: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization, Springer-Verlag, 1996, pp. 359–368. [cited at p. 43]

[18] 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. 22, 23, 24, 29, 48]

[19] HENRIK ANDERSSON (POSTDOC, UNIVERSITY OF BERGEN), *(Private e-mail correspondence)*, April 2009. [cited at p. 42]

[20] F. S. HILLIER AND G. J. LIEBERMAN, *Introduction to Operations Research*, McGraw-Hill Science/Engineering/Math, July 2004. [cited at p. 5]

[21] 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. 30, 44]

[22] 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. 30]

[23] N. KARMARKAR, *A new polynomial-time algorithm for linear programming*, in STOC '84: Proceedings of the sixteenth annual ACM symposium on Theory of computing, New York, NY, USA, 1984, ACM, pp. 302–311. [cited at p. 27]

[24] 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. 30, 56]

[25] 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. 16, 27]

[26] 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. 16]

[27] D. E. KNUTH, *Structured Programming with go to Statements*, ACM Computing Surveys, 6 (1974), pp. 261–301. [cited at p. 42]

[28] A. KOBERSTEIN, *The Dual Simplex Method — Techniques for a fast and stable implementation*, PhD thesis, University of Paderborn, 2005. [cited at p. 43]

[29] 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. 56, 58]

[30] I. MAROS, *Computational Techniques of the Simplex Method*, Kluwer Academic Publishers, Norwell, MA, USA, 2002. [cited at p. 24, 41, 42]

[31] B. MONIEN AND J. SCHULZE, *Parallel Sparse Cholesky Factorization*. [cited at p. 55]

[32] 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. 33]

[33] W. PRESS, S. TEUKOLSKY, W. VETTERLING, AND B. FLANNERY, *Numerical Recipes in C*, Cambridge University Press, 2nd ed., 1992. [cited at p. 28, 42]

[34] 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. 28, 42]

[35] J. SCHULZE, *Parallel Sparse Cholesky Factorization*. [cited at p. 55]

[36] 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. 34, 57]

[37] D. G. SPAMPINATO, *Linear Optimization with CUDA*. Fall project report, Norwegian University of Science and Technology, January 2009. [cited at p. 29, 56]

[38] R. J. VANDERBEI, *Linear Programming: Foundations and Extensions*, Springer, 2nd ed., 2001. [cited at p. 3, 11, 12, 13, 14, 17, 20, 21, 28, 39, 41]

[39] 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. 33]

[40] R. WUNDERLING, *Paralleler und objektorientierter Simplex-Algorithmus*, PhD thesis, Technische Universtät Berlin, Fachbereich Mathematik (G. Ziegler) and ZIB (M. Grötschel), December 1996. [cited at p. 29]

[41] G. YARMISH, *A Distributed Implementation of the Simplex Method*, PhD thesis, Poly-
      technic University, March 2001. [cited at p. 28, 40]

[42] G. YARMISH AND R. V. SLYKE, *retroLP, An Implementation of the Standard Sim-
      plex Method*, Tech. Rep. TR-CIS-2001-05, Polytechnic University, August 2001.
      [cited at p. 28, 42]

[43] E. A. YILDIRIM AND S. J. WRIGHT, *Warm-Start Strategies in Interior-Point Meth-
      ods for Linear Programming*, SIAM Journal on Optimization, 12 (2002), pp. 782–810.
      [cited at p. 17]

[44] K. YOSHII, *Time-stamp counter*.   http://www.mcs.anl.gov/~kazutomo/
      rdtsc.html. [cited at p. 50, 80]

Appendices

Appendix A

Code

A.1 **(RP)** Sequential standard simplex method for x86 and Cell/B.E.

(TODO: Defines) (TODO: Input format) (TODO: Describe each file: purpose, and if we have changed it)

It can be compiled in several versions:

- For x86, using `float`: run `g++ -O3 -Wall *.cpp`
  `-o standard_simplex_x86_float -DTYPE=float`

- For x86, using `double`: run `g++ -O3 -Wall *.cpp`
  `-o standard_simplex_x86_float -DTYPE=double`

- For x86, using GMP: run `g++ -O3 -Wall *.cpp`
  `-o standard_simplex_x86_gmp -DTYPE=mpq_class -lgmpxx -lgmp -DUSE_GMP`

- For Cell/B.E. (but uses only the PPE), using `float` and SIMD instructions: run `make`

- For Cell/B.E., using `float` and no SIMD instructions: edit the `Makefile` and remove `-DUSE_SIMD`, and run `make`

Note that the code highlighting package we use (`listings`) erroneously highglights the `vector` class from the Standard Template Library, thinking that it is the `vector` keyword for SIMD data types.

Matrix.h and **Matrix.cpp**

A class for representing dense matrices where the size of the physical memory buffer for each row is a multiple of 16 bytes, so that it supports SIMD operations internally. The actual matrix with can be anything.

```
#ifndef MATRIX_H
#define MATRIX_H

#include <iostream>
```

```cpp
#include <vector>
#include "gmpInterop.h"

class Matrix {
  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 &);
  ~Matrix();
  int getRows() { return rows; }
  int getCols() { return cols; }
  int getPhysicalCols() { return physicalCols; }

#ifdef VOLATILE_DATA
  // When this class is being used in the parallel standard simplex method for
  //     Cell, the data buffer must be marked as volatile because it will be the
  //     target of DMA transfers
  volatile
#endif
  TYPE * getData() { return data; }

  inline
#ifdef VOLATILE_DATA
  volatile
#endif
  TYPE & operator () (int r, int c) { return data[r * physicalCols + c]; }

  inline TYPE operator () (int r, int c) const { return data[r * physicalCols
      + c]; }
  void swapRows(int firstRow, int secondRow);
  void multiplyRow(int row, TYPE factor);
  void addRows(int sourceRow, int destinationRow, TYPE factor);
  void print(const std::vector<int> & basic, const std::vector<int> & nonbasic
      );
private:
  int rows;
  int cols;
  int physicalCols;
#ifdef VOLATILE_DATA
  volatile
#endif
  TYPE * data;
};

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

```cpp
// Used by addRows to increment one number by another, with support for two
    different "zeroing rules", controlled by defines:
// - set to zero if result is below ZEROING_RULE_EPSILON
// - set to zero if the ratios of each of the operands to the result is
    greater than ZEROING_RULE_RATIO
// - not defining either of these will simply result in a regular a += b
    operation
#ifdef VOLATILE_DATA
inline void incr(volatile TYPE & a, const TYPE & b) {
#else
inline void incr(TYPE & a, const TYPE & b) {
#endif
  #ifdef ZEROING_RULE_EPSILON
    a += b;
    if (ABS(a) <= ZEROING_RULE_EPSILON) {
      a = 0;
    }
  #else
    #ifdef ZEROING_RULE_RATIO
      TYPE result = a + b;
      if (result == 0 || (ABS(a / result) >= ZEROING_RULE_RATIO && ABS(b /
          result) >= ZEROING_RULE_RATIO)) {
        a = 0;
      }
      else {
        a = result;
      }
    #else
      a += b;
    #endif
  #endif
}

#endif
```

```cpp
#include "Matrix.h"
#define VECTOR_WIDTH (16 / sizeof(TYPE))
#define ALIGN_LOG2 4
#define ROUND_UP_MULTIPLE(x, m) (((x) + (m) - 1) / m * m) // Returns x rounded
    up to the nearest multiple of m

#ifdef __powerpc__
  // On Cell, we need these includes
  // Note: __powerpc__ should perhaps be replaced by something more Cell-
      specific in case one wants to compile this on a PowerPC that is not a
      Cell PPE
```

```
  #include <altivec.h>
  #include <libmisc.h>
#else
  #include <cstdlib>
  // On x86, there is no malloc_align and no need for it either
  #define malloc_align(size, alignment) malloc(size)
  #define free_align(buffer) free(buffer)
#endif

#ifdef USE_SIMD
  #ifdef USE_GMP
    #error "Cannot use GMP with SIMD operations"
  #endif
#endif

using namespace std;

Matrix::Matrix(int rows, int cols) {
  this->rows = rows;
  this->cols = cols;
  this->physicalCols = ROUND_UP_MULTIPLE(cols, VECTOR_WIDTH);
#ifdef USE_GMP
  // GMP's types are classes and require their constructors to be called, so
      we need to use new.
  // If necessary, it is possible to combine malloc_align and GMP through the
      use of 'placement new'.
  this->data = new mpq_class[rows * this->physicalCols];
#else
  this->data = (TYPE*)malloc_align(rows * this->physicalCols * sizeof(TYPE),
      ALIGN_LOG2);
#endif
  for (int i = 0; i < rows * this->physicalCols; ++i)
    this->data[i] = 0;
}

Matrix::Matrix(int rows, int cols, TYPE * data) {
  this->rows = rows;
  this->cols = cols;
  this->physicalCols = ROUND_UP_MULTIPLE(cols, VECTOR_WIDTH);
#ifdef USE_GMP
  this->data = new TYPE[rows * this->physicalCols];
#else
  this->data = (TYPE*)malloc_align(rows * this->physicalCols * sizeof(TYPE),
      ALIGN_LOG2);
#endif
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c)
```

```cpp
      this->data[r * this->physicalCols + c] = data[r * cols + c];
    for (int c = cols; c < this->physicalCols; ++c)
      this->data[r * this->physicalCols + c] = 0;
  }
}

Matrix::Matrix(int rows, int cols, bool identity) {
  this->rows = rows;
  this->cols = cols;
  this->physicalCols = ROUND_UP_MULTIPLE(cols, VECTOR_WIDTH);
#ifdef USE_GMP
  this->data = new TYPE[rows * this->physicalCols];
#else
  this->data = (TYPE*)malloc_align(rows * this->physicalCols * sizeof(TYPE),
      ALIGN_LOG2);
#endif
  for (int i = 0; i < rows * this->physicalCols; ++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->physicalCols = source.physicalCols;
#ifdef USE_GMP
  this->data = new TYPE[source.rows * source.physicalCols];
#else
  this->data = (TYPE*)malloc_align(source.rows * source.physicalCols * sizeof(
      TYPE), ALIGN_LOG2);
#endif
  for (int i = 0; i < source.rows * source.physicalCols; ++i)
    this->data[i] = source.data[i];
}

Matrix::~Matrix() {
#ifdef USE_GMP
  delete [] data;
#else
  free_align((float*)data);
#endif
}
```

```cpp
// Prints the entire matrix
ostream & operator << (ostream & out, const Matrix & matrix) {
  out << "=== " << matrix.rows << " x " << matrix.cols << " @ " << matrix.data
      << " ===" << endl;
  for (int r = 0; r < matrix.rows; ++r) {
    out << matrix(r, 0);
    for (int c = 1; c < matrix.cols; ++c)
      out << ' ' << matrix(r, c);
    out << endl;
  }
  out << "======" << endl;
  return out;
}

// A more advanced print operation that prints only nonzero entries, along
//   with variable names
void Matrix::print(const vector<int> & basic, const vector<int> & nonbasic) {
  cout << "=== " << rows << " x " << cols << " @ " << data << " ===" << endl;
  for (int r = 0; r < rows; ++r) {
    if (r == 0)
      cout << "z = ";
    else
      cout << "x" << basic[r - 1] << " = ";
    cout << (*this)(r, 0);
    for (int c = 1; c < cols; ++c)
      if ((*this)(r, c) != 0)
        cout << " " << (*this)(r, c) << "x" << nonbasic[c - 1];
    cout << endl;
  }
  cout << "======" << endl;
}

// No point in using SIMD here; this function is used very rarely
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;
  }
}




#ifdef USE_SIMD

  void Matrix::multiplyRow(int row, TYPE factor) {
```

```
    if (factor == 1) return;
    vector TYPE zero_v = (vector TYPE){0.0f, 0.0f, 0.0f, 0.0f};
    vector TYPE factor_v = (vector TYPE){factor, factor, factor, factor}; //
        Wanted to use vec_splat(vec_lde(0, &factor), 0) here, but might have
        misunderstood the syntax
    vector TYPE * data_v = (vector TYPE *)(data + row * physicalCols);
    for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
      data_v[j] = vec_madd(data_v[j], factor_v, zero_v);
    }
  }

  void Matrix::addRows(int sourceRow, int destinationRow, TYPE factor) {
    if (factor == 0) return;
    vector TYPE factor_v = (vector TYPE){factor, factor, factor, factor}; //
        As above
    vector TYPE * source_v = (vector TYPE *)(data + sourceRow * physicalCols);
    vector TYPE * destination_v = (vector TYPE *)(data + destinationRow *
        physicalCols);
    for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
      destination_v[j] = vec_madd(source_v[j], factor_v, destination_v[j]);
    }
  }

#else

  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);
    }
  }

#endif
```

mps.h and mps.cpp

An MPS parser. See notes in Appendix A.6 and Appendix A.6.

```
#ifndef MPS_H
#define MPS_H
```

```cpp
#include <iostream>
#include <vector>
#include <string>

std::vector<std::vector<TYPE> > parse(std::istream & input);
std::vector<std::vector<TYPE> > parse(const std::vector<std::string> & lines);

#endif
```

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <vector>
#include <stdexcept>
#include <sstream>
#include <cctype>
#include "gmpInterop.h"

using namespace std;

enum RowType {
  LESS_THAN,
  EQUAL_TO,
  GREATER_THAN,
  OBJECTIVE
};

string rowTypeLabels[] = {"L", "E", "G", "N"};

class Equation {
public:
  string label;
  RowType type;
  map<string, TYPE> values;
  int index;
  TYPE rhs;

  Equation(string label, string type, int index) {
    this->label = label;
    if (type == "L")
      this->type = LESS_THAN;
    else if (type == "E")
      this->type = EQUAL_TO;
    else if (type == "G")
      this->type = GREATER_THAN;
```

```cpp
    else if (type == "N")
      this->type = OBJECTIVE;
    else
      throw invalid_argument("");
    this->index = index;
    this->rhs = 0;
  }
};

string strip(string line) {
  int start = 0, end = (int)line.size() - 1;
  while (start < (int)line.size() && isspace(line[start])) ++start;
  while (end >= 0 && isspace(line[end])) --end;
  if (end < start)
    return "";
  else
    return line.substr(start, end - start + 1);
}

vector<string> split(string line) {
  stringstream ss(line);
  vector<string> items;
  string item;
  while (ss >> item) {
    items.push_back(item);
  }
  return items;
}

vector<vector<TYPE> > parse(const vector<string> & lines) {
  map<string, Equation *> equations;
  vector<string> columnLabels;
  map<string, int> columnIndices;
  vector<vector<TYPE> > tableau;
  unsigned int i = 0;
  int objectiveIndex = -1;
  while (i < lines.size()) {
    string line = lines[i];
    string header = strip(line);
    i++;
    if (line[0] == '*' || line[0] == ' ' || line.substr(0, 4) == "NAME") {
      continue;
    }
    else if (strip(line) == "ENDATA") {
      break;
    }
    else if (strip(line) == "ROWS") {
```

```cpp
    int rowIndex = 0;
    while (lines[i][0] == ' ') {
      vector<string> items = split(lines[i]);
      Equation * eqn = new Equation(strip(items[1]), strip(items[0]),
          rowIndex);
      if (eqn->type == OBJECTIVE)
        objectiveIndex = rowIndex;
      equations[eqn->label] = eqn;
      rowIndex++;
      i++;
    }
  }
  else if (strip(line) == "COLUMNS") {
    int columnIndex = -1;
    while (lines[i][0] == ' ') {
      vector<string> items = split(lines[i]);
      int lim = (items.size() == 5 ? 2 : 1);
      string columnLabel = strip(items[0]);
      if (columnIndices.find(columnLabel) == columnIndices.end()) {
        columnIndex++;
        columnLabels.push_back(columnLabel);
        columnIndices[columnLabel] = columnIndex;
      }
      for (int j = 0; j < lim; ++j) {
        string rowLabel = strip(items[1 + j * 2]);
        stringstream ss(strip(items[2 + j * 2]));
        TYPE value;
        readNumber(ss, value);
        equations[rowLabel]->values[columnLabel] = value;
      }
      i++;
    }
  }
  else if (strip(line) == "RHS") {
    while (lines[i][0] == ' ') {
      vector<string> items = split(lines[i]);
      int lim = (items.size() == 5 ? 2 : 1);
      for (int j = 0; j < lim; ++j) {
        string rowLabel = strip(items[1 + j * 2]);
        stringstream ss(strip(items[2 + j * 2]));
        TYPE value;
        readNumber(ss, value);
        equations[rowLabel]->rhs = value;
      }
      i++;
    }
  }
```

```cpp
    else {
      throw invalid_argument("");
    }
  }
  tableau.resize(1); // obj. func.
  for (map<string, Equation *>::iterator eqnIter = equations.begin(); eqnIter
      != equations.end(); ++eqnIter) {
    Equation * eqn = eqnIter->second;
    vector<TYPE> row(columnLabels.size() + 1, 0);
    bool any = false;
    row[0] = eqn->rhs;
    for (map<string, TYPE>::iterator colIter = eqn->values.begin(); colIter !=
        eqn->values.end(); ++colIter) {
      string colLabel = colIter->first;
      row[columnIndices[colLabel] + 1] = -eqn->values[colLabel];
      if (eqn->values[colLabel] != 0)
        any = true;
    }
    if (!any)
      continue;
    if (eqn->type == OBJECTIVE) {
      tableau[0] = row;
    }
    else {
      if (eqn->type == LESS_THAN || eqn->type == EQUAL_TO) {
        tableau.push_back(row);
      }
      if (eqn->type == GREATER_THAN || eqn->type == EQUAL_TO) {
        for (unsigned int j = 0; j < row.size(); ++j)
          row[j] = -row[j];
        tableau.push_back(row);
      }
    }
    delete eqn;
  }

  return tableau;
}

vector<vector<TYPE> > parse(istream & input) {
  string line;
  vector<string> lines;
  while (getline(input, line)) {
    lines.push_back(line);
  }
  return parse(lines);
}
```

gmpInterop.h and gmpInterop.cpp

Code for facilitating the use of GMP, whose classes must occasionally be treated differently from
primitive C++ types.

```cpp
#ifndef GMPINTEROP_H
#define GMPINTEROP_H

#include <iostream>
#ifdef USE_GMP
  #include <gmpxx.h>
  #define ABS(x) abs(x)
#else
  #include <cmath>
  #define ABS(x) fabs(x)
#endif


void readNumber(std::istream & in, TYPE & number);
void printNumber(const TYPE & number);
void printNumberFull(const TYPE & number);

#endif
```

```cpp
#ifdef USE_GMP
  #include <gmpxx.h>
#endif
#include "gmpInterop.h"
using namespace std;

#ifdef USE_GMP
// Only handles floats using regular notation (no 0x, no E) and integers.
// Accepts numbers < 1 without a leading zero.
// The string cannot be empty or contain any spaces.
static void parseRational(const string & str, mpq_class & number) {
  //bool negative = (str[0] == '-');
  size_t dotIndex = str.find('.');
  if (dotIndex == string::npos) {
    number = str;
    number.canonicalize();
    return;
  }
  string noDot = str.substr(0, dotIndex) + str.substr(dotIndex + 1);
//  size_t firstNonzeroIndex = noDot.find_first_not_of("-0");
//  if (firstNonzeroIndex == string::npos) {
//    number = 0;
//    return;
//  }
//  string noLeadingZeroes = negative ? "-" : "";
```

```cpp
//  noLeadingZeroes += noDot.substr(firstNonzeroIndex);
  mpq_class numerator(noDot, 10);
  int power = str.size() - dotIndex - 1;
  mpq_class denominator = 1;
  while (power--) {
    denominator *= 10;
  }
  number = numerator / denominator;
  number.canonicalize();
}
#endif


void readNumber(istream & in, TYPE & number) {
#ifdef USE_GMP
  string str;
  in >> str;
  parseRational(str, number);
#else
  in >> number;
#endif
}


void printNumber(const TYPE & number) {
#ifdef USE_GMP
  cout << number.get_d();
#else
  cout << number;
#endif
}


void printNumberFull(const TYPE & number) {
#ifdef USE_GMP
  cout << number.get_d() << " (approx); " << number << " (exact)";
#else
  cout << number;
#endif
}


/*
int main() {
  mpq_class a;
  parseRational("0.00001", a);
  cout << a << endl;
  parseRational("10.00001", a);
  cout << a << endl;
  parseRational("12.00000", a);
  cout << a << endl;
```

```
  parseRational(".02001", a);
  cout << a << endl;
  parseRational("-.5", a);
  cout << a << endl;
  parseRational("-0.00001", a);
  cout << a << endl;
  parseRational("-120", a);
  cout << a << endl;
  parseRational("-0.0", a);
  cout << a << endl;
  parseRational("-0.0", a);
  cout << a << endl;
  parseRational("0", a);
  cout << a << endl;
  parseRational("-0", a);
  cout << a << endl;
  parseRational("32.1234", a);
  cout << a << endl;
  return 0;
}
*/
```

timer.h

Precise timing module, taken from [44].

```
#ifndef TIMER_H
#define TIMER_H

// Thanks to Kazutomo Yoshii
// http://www.mcs.anl.gov/~kazutomo/rdtsc.html


#if defined(__i386__)

static __inline__ unsigned long long rdtsc(void)
{
  unsigned long long int x;
    __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
    return x;
}
#elif defined(__x86_64__)

static __inline__ unsigned long long rdtsc(void)
{
  unsigned hi, lo;
  __asm__ __volatile__ ("rdtsc" : "=a"(lo), "=d"(hi));
  return ( (unsigned long long)lo)|( ((unsigned long long)hi)<<32 );
```

```
}

#elif defined(__powerpc__)

static __inline__ unsigned long long rdtsc(void)
{
  unsigned long long int result=0;
  unsigned long int upper, lower,tmp;
  __asm__ volatile(
                "0:                      \n"
                "\tmftbu   %0            \n"
                "\tmftb    %1            \n"
                "\tmftbu   %2            \n"
                "\tcmpw    %2,%0         \n"
                "\tbne     0b            \n"
                : "=r"(upper),"=r"(lower),"=r"(tmp)
                );
  result = upper;
  result = result<<32;
  result = result|lower;

  return(result);
}

#else

#error "No tick counter is available!"

#endif


/*  $RCSfile: $    $Author: kazutomo $
 *  $Revision: 1.6 $  $Date: 2005/04/13 18:49:58 $
 */

#endif
```

TableauSimplex.h and TableauSimplex.cpp

Code for performing a simplex iteration (with pivoting).

```
#ifndef TALBEAUSIMPLEX_H
#define TALBEAUSIMPLEX_H

#include "Matrix.h"
#include <vector>
```

```cpp
enum SimplexResult {
  SUBOPTIMAL,
  OPTIMAL,
  INFEASIBLE_OR_UNBOUNDED
};

class TableauSimplex {
public:
  static SimplexResult iteration(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);
};

#endif
```

```cpp
#include "TableauSimplex.h"
#include "gmpInterop.h"
#include <cmath>
#include <vector>

using namespace std;

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

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

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

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

  // 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, j) > tableau(0,
        entering) || (tableau(0, j) == tableau(0, entering) && nonbasic[j - 1]
         < nonbasic[entering - 1])))
      entering = j;
  }
  if (entering == -1)
    return OPTIMAL;

  // Find leaving variable by searching the column of the entering variable
      and determine the strictest bound
  int leaving = -1;
  TYPE smallestRatio = -1; // Keep the compiler from complaining about
      uninitialised variables
  for (int i = 1; i <= m; ++i) {
    if (tableau(i, entering) >= 0)
      continue;
    TYPE ratio = -tableau(i, 0) / tableau(i, entering); // The "right hand
        side", tableau(i, 0), is always nonnegative, and we only get here if
        tableau(i, entering) is negative, so 'ratio' will be nonnegative
    if (leaving == -1 || ratio < smallestRatio || (ratio == smallestRatio &&
        basic[i - 1] < basic[leaving - 1])) {
      smallestRatio = ratio;
      leaving = i;
    }
  }
  if (leaving == -1)
    return INFEASIBLE_OR_UNBOUNDED;

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

  return SUBOPTIMAL;
}
```

simplex.cpp

Driver code — initiates input reading, handles the two phases, and initiates iterations.

```cpp
#include "Matrix.h"
```

```cpp
#include "TableauSimplex.h"
#include "mps.h"
#include <cmath>
#include <iostream>
#include <vector>
#include <cstdlib>
#include <fstream>
#include <cstring>
#include <ctime>
#include "../timer.h"

#define FEASIBILITY_THRESHOLD 1.0e-5 // abs(x0) must be below this value in
   order for the program to be declared feasible (only applies if Phase I is
   needed)

using namespace std;

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

  if (argc == 1)
    input = &cin;
  else {
    infile.open(argv[1]);
    input = &infile;
  }
  vector<vector<TYPE> > parsedTableau = parse(*input);
  rows = parsedTableau.size();
  cols = parsedTableau[0].size();
  Matrix A(rows, cols + 1);
  for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) {
      A(r, c) = parsedTableau[r][c];
    }
    if (r > 0 && A(r, 0) < 0) {
      initiallyFeasible = false;
    }
    A(r, cols) = 1;
  }

  unsigned long long startTime = rdtsc();

  std::vector<int> basic, nonbasic;
```

```cpp
// 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);

int numIterations = 0;
SimplexResult status = SUBOPTIMAL;

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 ((status = TableauSimplex::iteration(A, basic, nonbasic)) ==
      SUBOPTIMAL) {
    ++numIterations;
    if (print) A.print(basic, nonbasic);
    cout << numIterations << ": " << A(0, 0) << endl;
  }
  if (status == INFEASIBLE_OR_UNBOUNDED || ABS(A(0, 0)) >
      FEASIBILITY_THRESHOLD) {
    cout << "INFEASIBLE" << endl;
    return 0;
  }
  cout << "PHASE I COMPLETED" << endl;
  if (print) A.print(basic, nonbasic);
}

// Locate x0, which is expected to be nonbasic
int x0 = -1;
for (int i = 0; i < cols; ++i) {
  if (nonbasic[i] == 0) {
    x0 = i + 1;
```

```cpp
        nonbasic.erase(nonbasic.begin() + i);
        break;
      }
    }
    if (x0 == -1) {
      for (int j = 0; j < rows - 1; ++j) {
        if (basic[j] == 0) {
          x0 = j + 1;
          break;
        }
      }
      cout << "x0 is basic, and has value " << A(x0, 0) << " - terminating" <<
          endl;
      // If x0 is basic, but has value 0, it should be possible to continue by
          pivoting it out, but we haven't spent time on this since the program
          usually does not give the right answer anyway
      return 0;
    }

    // Even if there was no Phase I, we still copy the tableau - this should be
        avoided
    Matrix newTableau(rows, cols);
    for (int i = 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);
      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) {
          newTableau.addRows(i, 0, obj(0, basic[i - 1]));
        }
      }
    }

    if (print) newTableau.print(basic, nonbasic);
    cout << "Entering phase II" << endl;
    while ((status = TableauSimplex::iteration(newTableau, basic, nonbasic)) ==
        SUBOPTIMAL) {
      ++numIterations;
      if (print) newTableau.print(basic, nonbasic);
      cout << numIterations << ": " << newTableau(0, 0) << endl;
```

```cpp
  }
  if (status == INFEASIBLE_OR_UNBOUNDED) {
    cout << "UNBOUNDED" << endl;
    return 0;
  }
  cout << "OPTIMAL" << endl;
  cout << "Optimal value: " << newTableau(0, 0) << endl;
  cout << "Elapsed time (minus input parsing): " << rdtsc() - startTime <<
      endl;
  return 0;
}
```

Makefile

Makefile for compiling the Cell/B.E. versions. The file must be renamed to simply `Makefile` (unlike what it says under "List of code listings"); the code highlighting package does not seem to handle files without extensions.

```makefile
# Use comments to select if you want SIMD or not
#PROGRAM_ppu    := standard_simplex_ppe_float_serial
#CPPFLAGS_gcc = -DTYPE=float
PROGRAM_ppu   := standard_simplex_ppe_float_simd
CPPFLAGS_gcc  = -DTYPE=float -DUSE_SIMD




INCLUDE      := -I .

INSTALL_DIR   = /tmp
INSTALL_FILES = $(PROGRAM_ppu)

IMPORTS      = -lmisc



ifdef CELL_TOP
  include $(CELL_TOP)/buildutils/make.footer
else
  include ../../../../buildutils/make.footer
endif
```

A.2   (RP) Parallel standard simplex method for Cell/B.E.

This program also requires several files listed in the previous section: `gmpInterop.h`, `gmpInterop.cpp`, `Matrix.h`, `Matrix.cpp`, `mps.h`, `mps.cpp`, and `timer.h`. They (or a symlink to them, to avoid

unnecessary file duplication) should be placed in the top level source code folder of this program.

spu/`PartialMatrix.h` and spu/`PartialMatrix.cpp`

A class very similar to the `Matrix` class above, but with the property that it only contains a contiguous subset of the rows of the complete matrix. The idea is to give each SPE one part of the entire tableau. When addressing the `PartialMatrix`, all indices are relative to the original matrix.

```cpp
#ifndef PARTIALMATRIX_H
#define PARTIALMATRIX_H

class PartialMatrix {
public:
  PartialMatrix(int, int, int, int, int);
  PartialMatrix(int rows, int cols, int physicalCols, int startRow, int
      containedRows, float * data);
  PartialMatrix(int rows, int cols, int physicalCols, int startRow, int
      containedRows, bool identity);
  PartialMatrix(const PartialMatrix &);
  ~PartialMatrix();
  inline int getRows() const { return rows; }
  inline int getCols() const { return cols; }
  inline int getPhysicalCols() const { return physicalCols; }
  inline int getStartRow() const { return startRow; }
  inline int getContainedRows() const { return containedRows; }
  inline int getRowLimit() const { return startRow + containedRows; }
  inline volatile float * getData() { return data; }
  inline volatile float & operator () (int r, int c) { return data[(r -
      startRow) * physicalCols + c]; }
  inline float operator () (int r, int c) const { return data[(r - startRow) *
       physicalCols + c]; }
  void multiplyRow(int row, float factor);
  void addRows(float factor, int sourceRow, int destinationRow);
  void addRows(float factor, float * sourceRow, int destinationRow);
  void swapRows(int firstRow, int secondRow);
  void print();
private:
  void init(int rows, int cols, int physicalCols, int startRow, int
      containedRows);

  int rows;
  int cols;
  int physicalCols;
  int startRow;
  int containedRows;
  volatile float * data;
```

```
};

#endif
```

```cpp
#include "PartialMatrix.h"
#include <spu_intrinsics.h>
#include <libmisc.h>
#include "../types.h"
#include <stdio.h>

using namespace std;

void PartialMatrix::init(int rows, int cols, int physicalCols, int startRow,
    int containedRows) {
  this->rows = rows;
  this->cols = cols;
  this->physicalCols = ROUND_UP_MULTIPLE(cols, VECTOR_WIDTH);
  this->startRow = startRow;
  this->containedRows = containedRows;
  this->data = (float*)malloc_align(containedRows * physicalCols * sizeof(
      float), ALIGN_QUAD_LOG2);
}

PartialMatrix::PartialMatrix(int rows, int cols, int physicalCols, int
    startRow, int containedRows) {
  init(rows, cols, physicalCols, startRow, containedRows);
  for (int i = 0;  i < containedRows * physicalCols; ++i)
    this->data[i] = 0;
}

PartialMatrix::PartialMatrix(int rows, int cols, int physicalCols, int
    startRow, int containedRows, float * data) {
  init(rows, cols, physicalCols, startRow, containedRows);
  for (int r = 0; r < containedRows; ++r) {
    for (int c = 0; c < cols; ++c)
      this->data[r * physicalCols + c] = data[r * cols + c];
    for (int c = cols; c < physicalCols; ++c)
      this->data[r * physicalCols + c] = 0;
  }
}

PartialMatrix::PartialMatrix(int rows, int cols, int physicalCols, int
    startRow, int containedRows, bool identity) {
  init(rows, cols, physicalCols, startRow, containedRows);
  for (int i = 0;  i < containedRows * physicalCols; ++i)
    this->data[i] = 0;
  if (identity && rows == cols) {
```

```
    for (int i = 0; i < containedRows; ++i) {
      data[i * physicalCols + (startRow + i)] = 1;
    }
  }
}

PartialMatrix::PartialMatrix(const PartialMatrix & source) {
  init(source.rows, source.cols, source.physicalCols, source.startRow, source.
      containedRows);
  for (int i = 0;  i < containedRows * physicalCols; ++i)
    this->data[i] = source.data[i];
}

PartialMatrix::~PartialMatrix() {
  free_align((void *)data);
}

void PartialMatrix::print() {
  printf("=== %d x %d: %d rows: [%d, %d) ===\n", rows, cols, containedRows,
      startRow, getRowLimit());
  for (int r = startRow; r < getRowLimit(); ++r) {
    printf("%f", (*this)(r, 0));
    for (int c = 1; c < cols; ++c)
      printf(" %f", (*this)(r, c));
    printf("\n");
  }
  printf("======\n");
}

void PartialMatrix::multiplyRow(int row, float factor) {
  if (factor == 1) return;
  vector float zero_v = (vector float){0.0f, 0.0f, 0.0f, 0.0f};
  vector float factor_v = (vector float){factor, factor, factor, factor};
  vector float * data_v = (vector float *)(data + (row - startRow) *
      physicalCols);
  for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
    data_v[j] = spu_madd(data_v[j], factor_v, zero_v);
  }
}

void PartialMatrix::addRows(float factor, int sourceRow, int destinationRow) {
  if (factor == 0) return;
  vector float factor_v = (vector float){factor, factor, factor, factor};
  vector float * source_v = (vector float *)(data + (sourceRow - startRow) *
      physicalCols);
  vector float * destination_v = (vector float *)(data + (destinationRow -
      startRow) * physicalCols);
```

```
  for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
    destination_v[j] = spu_madd(source_v[j], factor_v, destination_v[j]);
  }
}

// Adds a multiple of a row that is not a part of this PartialMatrix to one of
//   the rows that are a part of it
void PartialMatrix::addRows(float factor, float * sourceRow, int
    destinationRow) {
  if (factor == 0) return;
  vector float factor_v = (vector float){factor, factor, factor, factor};
  vector float * source_v = (vector float *)sourceRow;
  vector float * destination_v = (vector float *)(data + (destinationRow -
      startRow) * physicalCols);
  for (int j = 0; j < physicalCols / VECTOR_WIDTH; ++j) {
    destination_v[j] = spu_madd(source_v[j], factor_v, destination_v[j]);
  }
}

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

spu/SpuTableauSimplex.h and spu/SpuTableauSimplex.cpp

Code for performing a simplex iteration (with pivoting) within an SPE and communicating with the PPE.

```
#ifndef SPUTABLEAUSIMPLEX_H
#define SPUTABLEAUSIMPLEX_H

#include "PartialMatrix.h"
#include "../types.h"

class SpuTableauSimplex {
public:
  static SimplexResult iterate(PartialMatrix & tableau, float * ppuPivotRow,
      uint tagId, int spuIndex);
  static void pivot(PartialMatrix & tableau, float * pivotRow, int leaving,
      int entering);
};
```

```
#endif
```

```cpp
#include "SpuTableauSimplex.h"
#include "../types.h"
#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include <libmisc.h>
#include <cstdio>

using namespace std;

void SpuTableauSimplex::pivot(PartialMatrix & tableau, float * pivotRow, int
   leaving, int entering) {
  float xFactor = pivotRow[entering];

  // Cancel out occurrences of the entering variable
  for (int i = tableau.getStartRow(); i < tableau.getRowLimit(); ++i) {
    if (i == leaving) continue;
    float factor = -tableau(i, entering) / xFactor;
    float savedColVal = tableau(i, entering);
    tableau.addRows(factor, pivotRow, i);
    tableau(i, entering) = savedColVal / xFactor;
  }

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

// Communications (* means "all";[^x] means "all except x"):
// Each communication end point is tagged in the code as "comm0" etc.
// 0. SPU0 -> PPU: entering variable, or optimality
// 1. PPU -> SPU[^0]: entering variable, or termination instruction
// 2. SPU* -> PPU: value and index of leaving variable, or unboundedness (both
//    determined locally; PPU determines global choice - let x be the spu that
//   'wins')
// 3. PPU -> SPU*: whether this spu 'won', or termination instruction
// 4. SPUx -> PPU: transfer pivot row; spu writes to (ppe reads from) mbox to
//   notify
// 5. PPU -> SPU[^x]: ppe writes to (spu reads from) mbox to sync; transfer
//   pivot row
SimplexResult SpuTableauSimplex::iterate(PartialMatrix & tableau, float *
   ppuPivotRow, uint tagId, int spuIndex) {
  int n = tableau.getCols() - 1;
  int entering = -1;
  volatile float * pivotRow;
```

```
static volatile float * incomingPivotRowBuffer = (float *)malloc_align(
    tableau.getPhysicalCols() * sizeof(float), ALIGN_QUAD_LOG2);

if (tableau.getStartRow() == 0) {
  // Find entering variable by searching the objective function (row 0) for
      a positive coefficient (disregard the constant in column 0)
  for (int j = 1; j <= n; ++j) {
    // We are not using Bland's rule here, since the SPEs do not maintain
        the basic/nonbasic lists (they could of course be arranged to do so)
    if (tableau(0, j) > 0 && (entering == -1 || tableau(0, j) > tableau(0,
        entering)))
      entering = j;
  }
  if (entering == -1) {
    spu_write_out_mbox(SIMPLEX_MBOX_OPTIMAL); //comm0
    return OPTIMAL;
  }
  else
    spu_write_out_mbox((uint)entering); //comm0
}
else {
  entering = (int)spu_read_in_mbox(); //comm1
  if ((uint)entering == SIMPLEX_MBOX_OPTIMAL)
    return OPTIMAL;
}

// Find leaving variable by searching the column of the entering variable
    and determine the strictest bound
int localLeaving = -1;
Value32 largestRatio;
int i = tableau.getStartRow();
if (i == 0) // Skip objective function row - this also handles the case
    where this partial matrix only contains the objective function, in which
     case SIMPLEX_MBOX_UNBOUNDED will be returned by SPE 0
  i = 1;
// We discovered a little too late that this part of the code is a little
    outdated - we are testing the inverse of the ratio that is described in
    the report, and therefore we must look for the largest ratio rather than
     the smallest. It is more cumbersome, but entirely equivalent to what is
     described in the report.
for (; i < tableau.getRowLimit(); ++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 (localLeaving == -1 || ratio > largestRatio.floatValue) {
    largestRatio.floatValue = ratio;
    localLeaving = i;
  }
}
//comm2
if (localLeaving == -1) {
  spu_write_out_mbox(SIMPLEX_MBOX_UNBOUNDED);
  spu_write_out_mbox((uint)-1);
}
else {
  spu_write_out_mbox(largestRatio.uintValue);
  spu_write_out_mbox((uint)localLeaving);
}

uint instruction = spu_read_in_mbox(); //comm3
if (instruction == SIMPLEX_MBOX_UNBOUNDED)
  return UNBOUNDED;

if (instruction == SIMPLEX_MBOX_LEAVING_IS_HERE) {
  //comm4 - pivot->ppu
  pivotRow = &tableau(localLeaving, 0);
  spu_mfcdma32((void *)pivotRow, (uint)ppuPivotRow, tableau.getPhysicalCols
      () * sizeof(float), tagId, MFC_PUT_CMD);
  (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
  spu_write_out_mbox(0U); //sync
}
else if (instruction == SIMPLEX_MBOX_LEAVING_IS_ELSEWHERE) {
  //comm5 - ppu->pivot
  pivotRow = incomingPivotRowBuffer;
  spu_read_in_mbox(); //sync
  spu_mfcdma32((void *)pivotRow, (uint)ppuPivotRow, tableau.getPhysicalCols
      () * sizeof(float), tagId, MFC_GET_CMD);
  (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
}
else {
  printf("Illegal instruction\n");
  return FAILURE;
}
```

```
    pivot(tableau, (float *)pivotRow, instruction ==
        SIMPLEX_MBOX_LEAVING_IS_HERE ? localLeaving : -1, entering);

    return SUBOPTIMAL;
}
```

`spu/spu.cpp`

Driver code for the SPE — transfers the initial tableau to the SPE, initiates the iterations, and transfers the final tableau back to the PPE.

```
#define SPE_CODE

#include <spu_intrinsics.h>
#include <spu_mfcio.h>
#include <libmisc.h>
#include <stdio.h>
# include "../types.h"
#include "PartialMatrix.h"
#include "SpuTableauSimplex.h"

#define MAX_DMA_SIZE 16384

volatile ParameterContext context ALIGNED_QUAD;

void safeDMA(volatile float * localStoreAddress, unsigned int effectiveAddress
    , unsigned int size, unsigned int tagId, unsigned int command) {
  int remainingSize = size;
  while (remainingSize > 0) {
    spu_mfcdma32(localStoreAddress, effectiveAddress, (remainingSize >
        MAX_DMA_SIZE ? MAX_DMA_SIZE : remainingSize), tagId, command);
    remainingSize -= MAX_DMA_SIZE;
    effectiveAddress += MAX_DMA_SIZE;
    localStoreAddress += MAX_DMA_SIZE / sizeof(float);
    (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);
  }
}

int main(unsigned long long spuId __attribute__ ((unused)), unsigned long long
    parameter) {
  uint tagId;
  if ((tagId = mfc_tag_reserve()) == MFC_TAG_INVALID) {
    printf("ERROR (%llx): unable to reserve a tag\n", spuId);
    return 1;
  }
  spu_writech(MFC_WrTagMask, -1);
```

```
  // Fetch the context and wait
  spu_mfcdma32((void *)&context, (uint)parameter, sizeof(ParameterContext),
      tagId, MFC_GET_CMD);
  (void)spu_mfcstat(MFC_TAG_UPDATE_ALL);

  int submatrixLength = context.numContainedRows * context.physicalCols;
  PartialMatrix matrix(context.rows, context.cols, context.physicalCols,
      context.startingRow, context.numContainedRows);
  safeDMA(matrix.getData(), (uint)(context.dataOrigin + context.startingRow *
      context.physicalCols), submatrixLength * sizeof(float), tagId,
      MFC_GETB_CMD); // MFC_GETB_CMD can be used for a barrier get, which will
       wait until previous puts have completed – http://publib.boulder.ibm.com
      /infocenter/systems/scope/syssw/index.jsp?topic=/eiccj/tutorial/
      cbet_3optimz.html

  while (SpuTableauSimplex::iterate(matrix, context.ppuPivotRow, tagId,
      context.spuIndex) == SUBOPTIMAL);

  safeDMA(matrix.getData(), (uint)(context.dataOrigin + context.startingRow *
      context.physicalCols), submatrixLength * sizeof(float), tagId,
      MFC_PUT_CMD);

  return 0;
}
```

types.h

Some structs, defines and enums that are needed by several of the other files.

```
#ifndef TYPES_H
#define TYPES_H

#define VECTOR_WIDTH 4
#define ALIGN_CACHE_WIDTH 128
#define ALIGN_CACHE_LOG2 7
#define ALIGN_QUAD_WIDTH 16
#define ALIGN_QUAD_LOG2 4
#define ALIGNED_CACHE __attribute__ ((aligned(ALIGN_CACHE_WIDTH)))
#define ALIGNED_QUAD __attribute__ ((aligned(ALIGN_QUAD_WIDTH)))
#define ROUND_UP_MULTIPLE(x, m) (((x) + (m) - 1) / (m) * (m))
#define PADDING(actualSize, alignmentWidth) char padding[ROUND_UP_MULTIPLE(
   actualSize, alignmentWidth) - (actualSize)] // Adds a char array to the
   end of a struct in order to make the struct take up a certain number of
   bytes – the first parameter is the combined size of the other struct
   memebers, and the second parameter is the byte boundary on which the
   struct size should be divisible.
```

```
// Sentinel values used when an SPE wants to notify the PPE that it did not
   find a good leaving or entering variable, and when the PPE wants to let
   the SPEs know who should perform the pivot. All of these are IEEE 754
   quiet NaNs, which means that they can't accidentally be interpreted as (or
    collide with) valid floating point numbers.
#define SIMPLEX_MBOX_OPTIMAL 0xffffffff
#define SIMPLEX_MBOX_UNBOUNDED 0xfffffffe
#define SIMPLEX_MBOX_LEAVING_IS_HERE 0xfffffffd
#define SIMPLEX_MBOX_LEAVING_IS_ELSEWHERE 0xfffffffc

//#define INFINITY (__builtin_inff())
#ifndef INFINITY
  #define INFINITY 3.4E38f
#endif

typedef unsigned int uint;

struct ParameterContext {
  int rows;
  int cols;
  int physicalCols;
  int startingRow;
  int numContainedRows;
  int spuIndex;
  float * dataOrigin;
  float * ppuPivotRow;
  PADDING(6 * sizeof(int) + 2 * sizeof(float*), ALIGN_QUAD_WIDTH);
};

// For interpreting a 32 bit pattern in different ways.
union Value32 {
  uint uintValue;
  int intValue;
  float floatValue;
};

enum SimplexResult {
  SUBOPTIMAL,
  OPTIMAL,
  UNBOUNDED,
  CYCLING,
  FAILURE
};

#endif
```

main.cpp

Driver code — initiates input reading, starts the SPE threads, and communicates with the threads.

```cpp
#define PPE_CODE

#include "Matrix.h"
#include "types.h"
#include "timer.h"
#include "mps.h"
#include <libspe2.h>
#include <libmisc.h>
#include <pthread.h>
#include <iostream>
#include <fstream>
using namespace std;


#define MAX_SPE_THREADS 8

extern spe_program_handle_t speProgramHandle; // Must match the name given in
    the SPE makefile

struct PpuThreadData {
  spe_context_ptr_t speContext;
  pthread_t pthread;
  void * argument;
};

SimplexResult simplexIteration(vector<int> & basic, vector<int> & nonbasic,
    int numSpeThreads, spe_context_ptr_t speContexts[]);

void * ppuPthreadFunction(void * argument) {
  PpuThreadData * data = (PpuThreadData *)argument;
  unsigned int entry = SPE_DEFAULT_ENTRY;
  if (spe_context_run(data->speContext, &entry, 0, data->argument, NULL, NULL)
      < 0) {
    perror("Failed running context");
    exit(1);
  }
  pthread_exit(NULL);
}

int main(int argc, char * argv[]) {
  int numSpeThreads;
  PpuThreadData data[MAX_SPE_THREADS];
  ParameterContext contexts[MAX_SPE_THREADS] ALIGNED_QUAD;
  spe_context_ptr_t speContexts[MAX_SPE_THREADS];
```

```cpp
if (argc < 2) {
  cout << "Must take MPS file name as parameter; the second argument is
      optional and specifies the number of SPEs" << endl;
  return 1;
}
ifstream infile(argv[1]);
vector<vector<float> > parsedTableau = parse(infile);
int rows = parsedTableau.size();
int cols = parsedTableau[0].size();
Matrix matrix(rows, cols);
for (int r = 0; r < rows; ++r) {
  for (int c = 0; c < cols; ++c) {
    matrix(r, c) = parsedTableau[r][c];
  }
  if (r > 0 && matrix(r, 0) < 0) {
    cout << "This prototype does only handle Phase II; must be run on
        problems that are initially feasible (the entire right hand side
        must be nonnegative)" << endl;
    return 1;
  }
}
volatile float * pivotRow = (volatile float *)malloc_align(matrix.
    getPhysicalCols() * sizeof(float), ALIGN_QUAD_LOG2);

unsigned long long startTime = rdtsc();

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

// Determine how many SPE threads we want
if (argc >= 3) {
  numSpeThreads = atoi(argv[2]);
}
else {
  numSpeThreads = spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1);
}
if (numSpeThreads < 1)
  numSpeThreads = 1;
if (numSpeThreads > MAX_SPE_THREADS)
  numSpeThreads = MAX_SPE_THREADS;
```

```cpp
if (numSpeThreads > matrix.getRows())
  numSpeThreads = matrix.getRows();
cout << "Using " << numSpeThreads << " SPEs" << endl;

// Create and start SPE threads
for (int i = 0; i < numSpeThreads; ++i) {
  contexts[i].rows = matrix.getRows();
  contexts[i].cols = matrix.getCols();
  contexts[i].physicalCols = matrix.getPhysicalCols();
  contexts[i].startingRow = matrix.getRows() / numSpeThreads * i;
  if (i == numSpeThreads - 1)
    contexts[i].numContainedRows = matrix.getRows() - contexts[i].
        startingRow;
  else
    contexts[i].numContainedRows = matrix.getRows() / numSpeThreads;
  contexts[i].spuIndex = i;
  contexts[i].dataOrigin = (float *)matrix.getData(); // Removing 'volatile'
      ness is ok here, since this pointer will just be used by the SPU to
      indicate the MFC target
  contexts[i].ppuPivotRow = (float *)pivotRow;

  if ((data[i].speContext = spe_context_create(0, NULL)) == NULL) {
    perror("Failed creating context");
    exit(1);
  }
  if (spe_program_load(data[i].speContext, &speProgramHandle)) {
    perror("Failed loading program");
    exit(1);
  }
  speContexts[i] = data[i].speContext;
  data[i].argument = &contexts[i];
  if (pthread_create(&data[i].pthread, NULL, &ppuPthreadFunction, &data[i]))
      {
    perror("Failed creating thread");
    return 1;
  }
}

int numIterations = 0;
while (simplexIteration(basic, nonbasic, numSpeThreads, speContexts) ==
    SUBOPTIMAL) {
//  cout << "Done with iteration " << ++numIterations << endl;
}

// Uncomment this if you want the list of basic/nonbasic variables to be
    printed
/*
```

```cpp
  cout << "Basic:";
  for (uint i = 0; i < basic.size(); ++i)
    cout << " " << basic[i];
  cout << "\nNonbasic:";
  for (uint i = 0; i < nonbasic.size(); ++i)
    cout << " " << nonbasic[i];
  cout << endl;
  */

  // Wait for SPEs to terminate
  for (int i = 0; i < numSpeThreads; ++i) {
    if (pthread_join(data[i].pthread, NULL)) {
      perror("Failed joining thread");
      exit(1);
    }
    if (spe_context_destroy(data[i].speContext) != 0) {
      perror("Failed destroying context");
      exit(1);
    }
  }

  cout << "The optimal solution is " << matrix(0, 0) << endl;
  cout << "Elapsed time (minus input parsing): " << rdtsc() - startTime <<
      endl;
  return 0;
}



// For explanation of communication tags, see spu/SpuTableauSimplex.cpp
SimplexResult simplexIteration(vector<int> & basic, vector<int> & nonbasic,
    int numSpeThreads, spe_context_ptr_t speContexts[]) {
  //comm0
  while (!spe_out_mbox_status(speContexts[0]));
  uint entering;
  spe_out_mbox_read(speContexts[0], &entering, 1);

  //comm1
  for (int i = 1; i < numSpeThreads; ++i) {
    spe_in_mbox_write(speContexts[i], &entering, 1, SPE_MBOX_ALL_BLOCKING); //
        entering may be SIMPLEX_MBOX_OPTIMAL, in which case SPE code will
        terminate
  }
  if (entering == SIMPLEX_MBOX_OPTIMAL) {
    return OPTIMAL;
  }
```

```cpp
    //comm2
    // The case where the first SPE only contains the objective function will
        not cause problems. That SPE will report SIMPLEX_MBOX_UNBOUNDED since it
         doesn't find a leaving variable, so it will be ignored unless no other
        SPEs find a leaving variable - in which case the entire problem is
        unbounded.
    while (!spe_out_mbox_status(speContexts[0]));
    float maxLeavingValue = 0; // Initialise with arbitrary value to get rid of
        warning
    int maxLeavingSpu = -1;
    int maxLeavingIndex = -1;
    for (int i = 0; i < numSpeThreads; ++i) {
      Value32 leavingValue, leavingIndex;
      while (!spe_out_mbox_status(speContexts[i]));
      spe_out_mbox_read(speContexts[i], &leavingValue.uintValue, 1);
      while (!spe_out_mbox_status(speContexts[i]));
      spe_out_mbox_read(speContexts[i], &leavingIndex.uintValue, 1);
      //cout << i << " reports " << leavingValue.floatValue << endl;
      if (leavingValue.uintValue != SIMPLEX_MBOX_UNBOUNDED && (maxLeavingSpu ==
          -1 || leavingValue.floatValue > maxLeavingValue)) {
        maxLeavingValue = leavingValue.floatValue;
        maxLeavingIndex = leavingIndex.intValue;
        maxLeavingSpu = i;
      }
    }

    //comm3
    for (int i = 0; i < numSpeThreads; ++i) {
      uint instruction;
      if (maxLeavingSpu == -1)
        instruction = SIMPLEX_MBOX_UNBOUNDED;
      else if (maxLeavingSpu == i)
        instruction = SIMPLEX_MBOX_LEAVING_IS_HERE;
      else
        instruction = SIMPLEX_MBOX_LEAVING_IS_ELSEWHERE;
      spe_in_mbox_write(speContexts[i], &instruction, 1, SPE_MBOX_ALL_BLOCKING);
    }
    if (maxLeavingSpu == -1) {
      cout << "Unbounded" << endl;
      return UNBOUNDED; //TODO: cleanup
    }

//  cout << "Pivoting: variable in row " << maxLeavingIndex << " (in SPU " <<
    maxLeavingSpu << ") leaves, variable in column " << entering << " enters"
    << endl;

    //comm4 - sync
```

```
  uint dummySyncValue;
  while (!spe_out_mbox_status(speContexts[maxLeavingSpu]));
  spe_out_mbox_read(speContexts[maxLeavingSpu], &dummySyncValue, 1);

  //comm5
  for (int i = 0; i < numSpeThreads; ++i) {
    if (maxLeavingSpu != i) {
      spe_in_mbox_write(speContexts[i], &dummySyncValue, 1,
        SPE_MBOX_ALL_BLOCKING);
    }
  }

  // Swap basic and nonbasic variables
  int tmp = basic[maxLeavingIndex - 1];
  basic[maxLeavingIndex - 1] = nonbasic[entering - 1];
  nonbasic[entering - 1] = tmp;

  return SUBOPTIMAL;
}
```

Makefiles

`Makefile` for SPE code (located in the `spe` folder):

```
# Target

PROGRAM_spu   := speProgramHandle # Must match the name of the
    spe_program_handle_t in main.cpp
LIBRARY_embed := matrix_spu.a

IMPORTS      = -lmisc



# buildutils/make.footer

ifdef CELL_TOP
  include $(CELL_TOP)/buildutils/make.footer
else
  include ../../../../buildutils/make.footer
endif
```

`Makefile` for PPE code:

```
# Subdirectories

DIRS        := spu
```

```
# Target

PROGRAM_ppu    := standard_simplex_parallel_float_simd


# Local Defines

INCLUDE      := -I .

INSTALL_DIR   = /tmp
INSTALL_FILES = $(PROGRAM_ppu)

IMPORTS      = spu/matrix_spu.a -lspe2 -lpthread -lmisc
CPPFLAGS_gcc  = -DTYPE=float -DVOLATILE_DATA


# buildutils/make.footer

ifdef CELL_TOP
  include $(CELL_TOP)/buildutils/make.footer
else
  include ../../../../buildutils/make.footer
endif
```

A.3 (RP) ASYNPLEX, C# prototype

Note that this program does not include an MPS parser; it only accepts input in the format described in (TODO:).

AsynplexProcess.cs

Base class for the four process types.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.CompilerServices;

namespace CS_asynplex {
  abstract class AsynplexProcess {
    public string name { get; private set; }

    public AsynplexProcess(string name) {
      this.name = name;
    }
```

```csharp
    protected void print(string message) {
      lock (Communication.printLock) {
        Console.WriteLine("{0}: {1}", name, message);
      }
    }

    protected void send(string receiver, string tag, object payload) {
      Communication.send(new Message(name, receiver, tag, payload));
    }

    protected Message receive() {
      return Communication.receive(name);
    }

    protected Message receive(string sender) {
      return Communication.receive(name, sender);
    }

    protected Message receive(string sender, string tag) {
      return Communication.receive(name, sender, tag);
    }

    public abstract void run();
  }
}
```

BasisChangeProcess.cs

ASYNPLEX basis change manager.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace CS_asynplex {
  class BasisChangeProcess : AsynplexProcess {
    int kb;

    public BasisChangeProcess()
      : base("R") {
    }

    public override void run() {
      Message message;
```

```
      kb = 0;
      while (!Program.Finished) {
        while ((message = receive()) == null) {
          if (Program.Finished)
            return;
        }
        Debug.Assert(message.tag == "I5->R1");
        int ki = (int)message.payload;
        string output = message.sender + " offers chuzr on basis " + ki + "...
            ";
        if (ki == kb) {
          send(message.sender, "R2->I6", true);
          ++kb;
          output += "accepted";
        }
        else {
          send(message.sender, "R3->I6", false);
          output += "refused";
        }
        print(output);
      }
    }
  }
}
```

ColumnSelectionProcess.cs

ASYNPLEX column selection manager.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;

namespace CS_asynplex {
  enum VariableStatus {
    BASIC,
    UNSELECTED,
    SELECTED,
    REJECTED
  }

  class VariableInfo {
    public VariableStatus status;
    public int basisWhereStatusChanged;
```

```csharp
  public VariableInfo(VariableStatus status) {
    this.status = status;
    this.basisWhereStatusChanged = -1;
  }
}

class QueuedVariableRequest {
  public int unattractiveVariable;
  public int basis;
  public string processName;
  public string tagToSend;
}

class ColumnSelectionProcess : AsynplexProcess {
  //List<bool> selected = new List<bool>();
  List<VariableInfo> variables = new List<VariableInfo>();
  Queue<int> attractive = new Queue<int>();
  Queue<QueuedVariableRequest> queuedVariableRequests = new Queue<
      QueuedVariableRequest>();
  int m;
  int n;
  int kc;

  public ColumnSelectionProcess(int m, int n)
    : base("C") {
    this.m = m;
    this.n = n;
    this.kc = 0;
    this.variables.Add(null); // 1-based array; index 0 not used - TODO:
        This might not be the case in Phase I in Vanderbei?
    for (int i = 0; i < n; ++i) // nonbasic
      this.variables.Add(new VariableInfo(VariableStatus.UNSELECTED));
    for (int i = 0; i < m; ++i) // basic
      this.variables.Add(new VariableInfo(VariableStatus.BASIC));
  }

  private void installAttractiveCandidates(Message message) {
    AttractiveCandidatesMessage msg = (AttractiveCandidatesMessage)message.
        payload; // assumed to be sorted by attractiveness. index 0 is ki
    Debug.Assert(msg.attractiveVariables.Length >= 1);
    print("received " + msg.attractiveVariables.Length + " candidates from "
        + message.sender + ", which is at basis " + msg.basisNumber + ";
        currently at basis " + kc);
    if (msg.basisNumber > kc) {
      int installCount = 0;
      attractive = new Queue<int>();
      foreach (int cand in msg.attractiveVariables) {
```

```
      if (!((variables[cand].status == VariableStatus.SELECTED ||
          variables[cand].status == VariableStatus.REJECTED) && variables[
          cand].basisWhereStatusChanged >= msg.basisNumber)) {
        ++installCount;
        attractive.Enqueue(cand);
        variables[cand].status = VariableStatus.UNSELECTED;
        variables[cand].basisWhereStatusChanged = msg.basisNumber;
      }
      else {
        print("rejected x" + cand + " because it got " + variables[cand].
            status + " at basis " + variables[cand].
            basisWhereStatusChanged);
      }
    }
    print("Installed " + installCount + " candidates");
    kc = msg.basisNumber;
  }
}

private void waitForAndInstallAttractiveCandidates() {
  Message message;
  while ((message = receive("I*", "I9->C2")) == null) ;
  installAttractiveCandidates(message);
}

public override void run() {
  Message message;
  //waitForAndInstallAttractiveCandidates();
  while (!Program.Finished) {
    while (queuedVariableRequests.Count > 0 && attractive.Count > 0) {
      if (Program.Finished)
        return;
      int var = attractive.Dequeue();
      QueuedVariableRequest req = queuedVariableRequests.Dequeue();
      Debug.Assert(variables[var].status == VariableStatus.UNSELECTED);
      variables[var].status = VariableStatus.SELECTED;
      variables[var].basisWhereStatusChanged = kc;
      print("x" + var + " gets SELECTED (C5) at basis " + kc);
      send(req.processName, req.tagToSend, var);
    }
    while ((message = receive()) == null) {
      if (Program.Finished)
        return;
    }
    Debug.Assert(message.sender[0] == 'I');
    if (message.tag == "I8->C1") {
      int[] enteringLeavingBasis = (int[])message.payload;
```

```csharp
            variables[enteringLeavingBasis[0]].status = VariableStatus.BASIC;
            variables[enteringLeavingBasis[0]].basisWhereStatusChanged =
                enteringLeavingBasis[2];
            variables[enteringLeavingBasis[1]].status = VariableStatus.
                UNSELECTED;
            variables[enteringLeavingBasis[1]].basisWhereStatusChanged =
                enteringLeavingBasis[2];
            print("x" + enteringLeavingBasis[1] + " left basis " +
                enteringLeavingBasis[2] + " (by " + message.sender + ") and got
                unselected");
          }
          else if (message.tag == "I9->C2") {
            installAttractiveCandidates(message);
            //queuedVariableRequests.Enqueue(new QueuedVariableRequest() {
                processName = message.sender, tagToSend = "C3->I11" });
            int var = attractive.Dequeue();
            Debug.Assert(variables[var].status == VariableStatus.UNSELECTED);
            variables[var].status = VariableStatus.SELECTED;
            variables[var].basisWhereStatusChanged = kc;
            print("x" + var + " gets SELECTED (C3) at basis " + kc);
            send(message.sender, "C3->I11", var);
          }
          else if (message.tag == "I4->C4") {
            int[] rejectedAndBasis = (int[])message.payload;
            if (rejectedAndBasis[0] != -1) {
              variables[rejectedAndBasis[0]].status = VariableStatus.REJECTED;
              variables[rejectedAndBasis[0]].basisWhereStatusChanged =
                  rejectedAndBasis[1];
              print("x" + rejectedAndBasis[0] + " gets REJECTED (C5) at basis "
                  + rejectedAndBasis[1]);
            }
            queuedVariableRequests.Enqueue(new QueuedVariableRequest() {
                processName = message.sender, tagToSend = "C5->I11" });
          }
          else {
            Debug.Assert(false, "C: Unexpected tag '" + message.tag + "'");
          }
        }
      }
    }
  }
}
```

Communication.cs

A simple message passing system.

```csharp
using System;
```

```csharp
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.CompilerServices;

namespace CS_asynplex
{
  class Message {
    public string sender { get; private set; }
    public string receiver { get; private set; }
    public string tag { get; private set; }
    public object payload { get; private set; }

    public Message(string sender, string receiver, string tag, object payload)
        {
      this.sender = sender;
      this.receiver = receiver;
      this.tag = tag;
      this.payload = payload;
    }
  }



    class Communication
    {
    static List<Message> messages = new List<Message>();
    static bool printSends = true;
    static bool printReceives = true;
    static object messageLock = new object();
    public static readonly object printLock = new object();

    public static void send(Message message) {
      lock (messageLock) {
        messages.Add(message);
      }
      if (printSends) {
        lock (printLock) {
          Console.ForegroundColor = ConsoleColor.DarkGray;
          Console.WriteLine("COMM: {0} sends to {1} (tag: '{2}', msg type:
              {3}) (queue: {4})", message.sender, message.receiver, message.
              tag, message.payload == null ? "(null)" : message.payload.
              GetType().ToString(), messages.Count);
          Console.ForegroundColor = ConsoleColor.Gray;
        }
      }
    }
```

```csharp
public static Message receive(string receiver) {
  return receive(receiver, null, null);
}

public static Message receive(string receiver, string sender) {
  return receive(receiver, sender, null);
}

public static Message receive(string receiver, string sender, string tag)
    {
  lock (messageLock) {
    for (int i = 0; i < messages.Count; ++i) {
      if (messages[i].receiver == receiver && (sender == null || messages[
          i].sender == sender || (sender == "I*" && messages[i].sender[0]
          == 'I')) && (tag == null || messages[i].tag == tag)) {
        Message message = messages[i];
        messages.RemoveAt(i);
        if (printReceives) {
          lock (printLock) {
            Console.ForegroundColor = ConsoleColor.DarkGray;
            Console.WriteLine("COMM: {0} receives from {1} (tag: '{2}',
                msg type: {3}) (queue: {4})", message.receiver, message.
                sender, message.tag, message.payload == null ? "(null)" :
                message.payload.GetType().ToString(), messages.Count);
            Console.ForegroundColor = ConsoleColor.Gray;
          }
        }
        return message;
      }
    }
  }
  return null;
}

public static void printMessages() {
  lock (messageLock) {
    lock (printLock) {
      Console.Write("Remaining messages (some messages may subsequently be
          read before all threads terminate):");
      foreach (Message msg in messages) {
        Console.Write(" " + msg.tag);
      }
      Console.WriteLine();
    }
  }
}
```

```
    }
}
```

InversionProcess.cs

ASYNPLEX invert processor.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.Threading;

namespace CS_asynplex {
  class InverseMatrixMessage {
    public int basisNumber;
    public Matrix inverse;
  }

  class InversionProcess : AsynplexProcess {
    private Matrix B, N;
    int numIterationProcs;
    int lastReinversionBasis = 0;
    int basisNumber = 0;

    public InversionProcess(Matrix N, int numIterationProcs)
      : base("V") {
      this.N = new Matrix(N);
      this.B = new Matrix(N.Rows, N.Rows, true, "B");
      this.numIterationProcs = numIterationProcs;
    }

    public override void run() {
      Message message;

      while (!Program.Finished) {
        // V1 <- I8
        while ((message = receive()) != null || basisNumber ==
            lastReinversionBasis) {
          if (Program.Finished)
            return;
          if (message == null) {
            Thread.Sleep(10);
            continue;
          }
          Debug.Assert(message.tag == "I8->V1");
```

```csharp
            int[] swap = (int[])message.payload;
            int entering = swap[0];
            int leaving = swap[1];
            Matrix tempCol = B.getColumn(leaving);
            B.setColumn(leaving, N.getColumn(entering));
            N.setColumn(entering, tempCol);
            ++basisNumber;
            print(message.sender + " says to let " + leaving + " leave and " +
                entering + " enter; now at basis " + basisNumber);
          }
          if (Program.Finished)
            return;

          print("Calculating inverse for basis " + basisNumber);
          Matrix BInverse = B.invert();
          lastReinversionBasis = basisNumber;
          //Thread.Sleep(8000);//TODO:remove
          print("Done with inverse for basis " + basisNumber);

          // V2 -> I1
          for (int i = 0; i < numIterationProcs; ++i) {
            //TODO: Must also tell what basis this is for
            InverseMatrixMessage invMsg = new InverseMatrixMessage();
            invMsg.basisNumber = basisNumber;
            invMsg.inverse = BInverse;
            send("I" + i, "V2->I1", invMsg);
          }
        }
      }
    }
}
```

IterationProcess.cs

ASYNPLEX iteration process.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.Threading;

namespace CS_asynplex {
  struct AttractiveCandidate {
    public float reducedCost;
    public int index;
```

```csharp
  public int column;
}

class AttractiveCandidatesMessage {
  public int basisNumber;
  public int[] attractiveVariables;
}

class BasisChangeMessage {
  public int enteringVar;
  public int leavingVar;
  public Matrix stepDirection;
  public float stepLength;
}

class IterationProcess : AsynplexProcess, IComparer<AttractiveCandidate> {
  int processIndex;
  int numIterationProcesses;
  int m, n;
  int ki;
  int currentEtaBasis = 0;
  const int reinversionPeriod = 5;
  Matrix N, B, originalBInverse;
  Matrix cNonbasic;
  Matrix cBasic;
  Matrix basicValues;
  Matrix origRhs;
  int[] nonbasicVars;
  int[] basicVars;
  bool[] basicStatus;
  int[] position;
  PrintControl printControl;
  const float EPS = 0.0000005f;
  List<Matrix> etaFile = new List<Matrix>();

  public void swap(ref int a, ref int b) {
    int temp = a;
    a = b;
    b = temp;
  }

  public IterationProcess(int processIndex, int numIterationProcesses, int m
      , int n, Matrix N, Matrix basicValues, Matrix cNonbasic)
    : base("I" + processIndex) {
    this.processIndex = processIndex;
    this.numIterationProcesses = numIterationProcesses;
    this.m = m;
```

```csharp
    this.n = n;
    this.N = new Matrix(N);
    this.cNonbasic = new Matrix(cNonbasic);
    this.origRhs = new Matrix(basicValues);
    this.basicValues = new Matrix(basicValues);

    this.B = new Matrix(N.Rows, N.Rows, true, "B");
    this.originalBInverse = new Matrix(N.Rows, N.Rows, true, "B^I");
    this.cBasic = new Matrix(m, 1, "cBasic");
    this.nonbasicVars = new int[n];
    this.basicVars = new int[m];
    this.basicStatus = new bool[m + n + 1]; //1-indexed
    this.position = new int[m + n + 1]; //1-indexed

    for (int i = 1; i <= n; ++i) {
      this.nonbasicVars[i - 1] = i;
      this.basicStatus[i] = false;
      this.position[i] = i - 1;
    }
    for (int i = n + 1; i <= n + m; ++i) {
      this.basicVars[i - n - 1] = i;
      this.basicStatus[i] = true;
      this.position[i] = i - n - 1;
    }

    printControl = new PrintControl() { iterationCount = false,
        objectiveValue = true, pivotResults = true };
}

private static bool equalsEpsilon(float a, float b) {
  return Math.Abs(a - b) < EPS;
}

public int Compare(AttractiveCandidate a, AttractiveCandidate b) {
  if (equalsEpsilon(a.reducedCost, b.reducedCost))
    return a.index.CompareTo(b.index);
  return b.reducedCost.CompareTo(a.reducedCost);
}

private void BTRAN_calcPiTranspose(out Matrix piTranspose) {
  piTranspose = cBasic.transpose() * calcBInverseFromEtaFile(); // \pi^T
  piTranspose.Name = "pi^T";
  if (printControl.piTranspose) piTranspose.print();
}

private void PRICE_calcReducedCosts(Matrix piTranspose, out Matrix
    reducedCosts) {
```

```csharp
    reducedCosts = cNonbasic.transpose() - piTranspose * N; // \hat{c}_N^T
    reducedCosts.Name = "reduced costs";
    if (printControl.reducedCosts) reducedCosts.print();
}

private void CHUZC_findAttractiveEntering(Matrix reducedCosts, out int
    enteringCol, out int enteringVar, out List<AttractiveCandidate>
    attractive) {
    attractive = new List<AttractiveCandidate>();
    for (int c = 0; c < reducedCosts.Cols; ++c) {
      if (reducedCosts[0, c] > 0) {
        attractive.Add(new AttractiveCandidate() { reducedCost =
            reducedCosts[0, c], index = nonbasicVars[c], column = c });
      }
    }
    attractive.Sort(this);

    if (printControl.pivotDecisions) {
      string output = "attractive candidates:";
      foreach (AttractiveCandidate cand in attractive) {
        output += " x" + cand.index + "(" + cand.reducedCost + ")";
      }
      print(output);
    }

    if (attractive.Count == 0) {
      print("Optimal");
      enteringCol = -1;
      enteringVar = -1;
      return;
    }
    else if (attractive.Count <= processIndex) {
      enteringCol = -1;
      enteringVar = -1;
      //throw new NotImplementedException();
      //TODO: What to do here?
    }
    else {
      enteringCol = attractive[processIndex].column;
      enteringVar = attractive[processIndex].index;
    }

    if (printControl.pivotResults) print("Entering variable: x" +
        enteringVar + " (column " + enteringCol + ")");
}

private void CHUZR_chooseLeaving(Matrix stepDirection, out int leavingRow,
```

```csharp
     out int leavingVar, out float stepLength) {
   leavingRow = -1; // p
   float largestRatio = 0; // step length (\alpha)
   for (int r = 0; r < basicValues.Rows; ++r) {
     float ratio;
     if (stepDirection[r, 0] == 0 && basicValues[r, 0] == 0)
       ratio = 0;
     else
       ratio = stepDirection[r, 0] / basicValues[r, 0];
     if (ratio > 0 && (leavingRow == -1 || ratio > largestRatio || (ratio
         == largestRatio && basicVars[r] < basicVars[leavingRow]))) {
       if (printControl.pivotDecisions) print("Choosing " + (r+1) + " over
           " + (leavingRow+1) + " to leave; ratio is " + ratio);
       leavingRow = r;
       largestRatio = ratio;
     }
   }
   if (leavingRow == -1) {
     print("Unbounded");
     leavingVar = -1;
     stepLength = float.PositiveInfinity;
     return;
   }
   stepLength = 1 / largestRatio;
   leavingVar = basicVars[leavingRow];
   if (printControl.pivotResults) print("Leaving variable: x" + leavingVar
       + " (row " + leavingRow + ")");
   if (printControl.stepLength) print("Step length: " + stepLength);
}

private Matrix calcBInverseFromEtaFile() {
   Matrix result = originalBInverse;
   foreach (Matrix eta in etaFile) {
     result = eta * result;
   }
   if (etaFile.Count > 0 && printControl.inverseDiscrepancy) {
     float max = 0;
     Matrix discrepancy = result - B.invert();
     for (int r = 0; r < discrepancy.Rows; ++r) {
       for (int c = 0; c < discrepancy.Cols; ++c) {
         float absVal = Math.Abs(discrepancy[r, c]);
         if (absVal > max)
           max = absVal;
       }
     }
     Console.WriteLine("Largest discrepancy: {0}", max);
   }
```

```
    return result;
  }

 private void UPDATE_BASIS(int enteringCol, int leavingRow, Matrix
    stepDirection) {
   Matrix tempM = N.getColumn(enteringCol);
   N.setColumn(enteringCol, B.getColumn(leavingRow));
   B.setColumn(leavingRow, tempM);

   float dxi = stepDirection[leavingRow, 0];
   if (dxi == 0) throw new Exception();
   Matrix etaMatrix = new Matrix(B.Rows, B.Cols, true, "Eta" + etaFile.
      Count);
   for (int r = 0; r < etaMatrix.Rows; ++r) {
     if (r == leavingRow)
       etaMatrix[r, leavingRow] = 1 / dxi;
     else
       etaMatrix[r, leavingRow] = -stepDirection[r, 0] / dxi;
   }
   etaFile.Add(etaMatrix);
   if (printControl.etaFileSize) Console.WriteLine("{0} eta matrices",
      etaFile.Count);

   int enteringVar = nonbasicVars[enteringCol];
   int leavingVar = basicVars[leavingRow];

   float tempF = cNonbasic[enteringCol, 0];
   cNonbasic[enteringCol, 0] = cBasic[leavingRow, 0];
   cBasic[leavingRow, 0] = tempF;

   int tempI = nonbasicVars[enteringCol];
   nonbasicVars[enteringCol] = basicVars[leavingRow];
   basicVars[leavingRow] = tempI;

   basicStatus[enteringVar] = true;
   basicStatus[leavingVar] = false;
   tempI = position[enteringVar];
   position[enteringVar] = position[leavingVar];
   position[leavingVar] = tempI;

   if (printControl.variableLists) {
     Console.Write("Basic:    ");
     for (int i = 0; i < m; ++i)
       Console.Write(basicVars[i] + " ");
     Console.Write("\nNonbasic: ");
     for (int i = 0; i < n; ++i)
       Console.Write(nonbasicVars[i] + " ");
```

```csharp
      Console.WriteLine();
    }
    if (printControl.iterationCount) print("\nCompleted iteration " + ki);
    if (printControl.iterationSeparator) print("\n\n");
}

private void UPRHS(int leavingRow, float stepLength, Matrix stepDirection)
    {
  basicValues -= stepLength * stepDirection;
  basicValues[leavingRow, 0] = stepLength;
  //basicValues = calcBInverseFromEtaFile() * origRhs;
  //basicValues.Name = "b";
  if (printControl.objectiveValue) {
    Matrix objectiveValue = cBasic.transpose() * basicValues;
    Debug.Assert(objectiveValue[0, 0] >= 0);
    print("Objective value: " + objectiveValue[0, 0]);
  }
}

private void applyIncomingBasisChanges() {
  Message message;
  while ((message = receive("I*", "I7->I2I3I10")) != null) {
    Debug.Assert(message.tag == "I7->I2I3I10");
    BasisChangeMessage msg = (BasisChangeMessage)message.payload;
    UPDATE_BASIS(position[msg.enteringVar], position[msg.leavingVar], msg.
        stepDirection);
    UPRHS(position[msg.leavingVar], msg.stepLength, msg.stepDirection);
    ++ki;
  }
}

public override void run() {
  Message message;
  ki = 0;

  Matrix piTranspose;
  BTRAN_calcPiTranspose(out piTranspose);

  Matrix reducedCosts;
  PRICE_calcReducedCosts(piTranspose, out reducedCosts);

  int enteringCol, enteringVar;
  List<AttractiveCandidate> attractive;
  CHUZC_findAttractiveEntering(reducedCosts, out enteringCol, out
      enteringVar, out attractive);

  while (!Program.Finished) {
```

```
while ((message = receive("V")) != null) {
  Debug.Assert(message.tag == "V2->I1");
  InverseMatrixMessage invMsg = (InverseMatrixMessage)message.payload;
  if (invMsg.basisNumber <= currentEtaBasis) { // Skip outdated
      matrices
    print("Got outdated B^I for basis " + invMsg.basisNumber + ";
        currently at basis " + currentEtaBasis + " / " + ki);
    continue;
  }
  if (invMsg.basisNumber > ki) {
    print("Got too new B^I for basis " + invMsg.basisNumber + ";
        currently at basis " + currentEtaBasis + " / " + ki);
    continue;
  }
  originalBInverse = invMsg.inverse;
  int removeCount = invMsg.basisNumber - currentEtaBasis;
  Debug.Assert(etaFile.Count >= removeCount);
  etaFile.RemoveRange(0, removeCount);
  currentEtaBasis += removeCount;
  print("Got new B^I for basis " + currentEtaBasis + "; eta file size
      is " + etaFile.Count);
}
applyIncomingBasisChanges();

//FTRAN
Matrix stepDirection = null;
if (enteringCol != -1) {
  stepDirection = calcBInverseFromEtaFile() * N.getColumn(enteringCol)
      ; // \hat{a}_q
  stepDirection.Name = "step direction";
  if (printControl.stepDirection) stepDirection.print();
}

bool keepGoing;
do {
  keepGoing = false;
  applyIncomingBasisChanges();

  //FTRAN
  if (enteringCol != -1) {
    stepDirection = calcBInverseFromEtaFile() * N.getColumn(
        enteringCol); // \hat{a}_q
    stepDirection.Name = "step direction";
    if (printControl.stepDirection) stepDirection.print();
  }

  Debug.Assert((enteringVar == -1 && enteringCol == -1) || enteringCol
```

```
              == position[enteringVar]);
    if (enteringVar == -1 || basicStatus[enteringVar] || reducedCosts[0,
        enteringCol] < 0) { //TODO: The pseudocode uses < - why perform
        a zero-advance pivot?
      send("C", "I4->C4", new int[]{enteringVar, ki});
    }
    else {
      send("R", "I5->R1", ki);
      while ((message = receive("R")) == null) {
        if (Program.Finished)
          return;
        Thread.Sleep(10);
      }
      Debug.Assert(message.tag == "R2->I6" || message.tag == "R3->I6");
      bool accepted = (bool)message.payload;
      if (accepted) {
        int leavingRow, leavingVar;
        float stepLength;
        CHUZR_chooseLeaving(stepDirection, out leavingRow, out
            leavingVar, out stepLength);
        if (leavingRow == -1) {
          Console.WriteLine("Unbounded");
          Program.Finished = true;
          Console.ReadLine();
          return;
        }
        for (int i = 0; i < numIterationProcesses; ++i) {
          if (i == processIndex) continue;
          BasisChangeMessage basisChangeMessage = new BasisChangeMessage
              () {enteringVar = enteringVar, leavingVar = leavingVar,
              stepDirection = stepDirection, stepLength = stepLength};
          send("I" + i, "I7->I2I3I10", basisChangeMessage);
        }
        send("V", "I8->V1", new int[] { enteringCol, leavingRow, });
        send("C", "I8->C1", new int[] { enteringVar, leavingVar, ki });
        UPDATE_BASIS(enteringCol, leavingRow, stepDirection);
        UPRHS(leavingRow, stepLength, stepDirection);
        ++ki;
        BTRAN_calcPiTranspose(out piTranspose);
        PRICE_calcReducedCosts(piTranspose, out reducedCosts);
        CHUZC_findAttractiveEntering(reducedCosts, out enteringCol, out
            enteringVar, out attractive);
        if (attractive.Count == 0) {
          Console.WriteLine("Optimal");
          Program.Finished = true;
          Communication.printMessages();
          Console.ReadLine();
```

```
              return;
          }
          AttractiveCandidatesMessage msg = new
              AttractiveCandidatesMessage();
          msg.basisNumber = ki;
          msg.attractiveVariables = new int[attractive.Count];
          for (int i = 0; i < attractive.Count; ++i) {
            msg.attractiveVariables[i] = attractive[i].index;
            print(attractive[i].index + " is attractive");
          }
          send("C", "I9->C2", msg);
        }
        else {
          while ((message = receive("I*", "I7->I10")) == null) {
            if (Program.Finished)
                return;
            Thread.Sleep(10);
          }
          Debug.Assert(message.tag == "I7->I10", "Expected I7->I10, got "
              + message.tag);
          throw new NotImplementedException();
          //TODO: Apply basis change
          ++ki;
          print("Keep going");
          keepGoing = true;
        }
      }
    } while (keepGoing);

    while ((message = receive("C")) == null) {
      if (Program.Finished)
        return;
      Thread.Sleep(10);
    }
    Debug.Assert(message.tag == "C3->I11" || message.tag == "C5->I11");
    applyIncomingBasisChanges();
    enteringVar = (int)message.payload;
    enteringCol = position[enteringVar];
    print("got suggested entering variable: x" + enteringVar);
  }
  }
 }
}
```

Matrix.cs

A class for representing dense matrices and performing linear algebra operations on them.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CS_asynplex {
  class Matrix {
    private int rows;
    private int cols;
    private float[] data;
    private string name;

    public int Rows { get { return rows; } }
    public int Cols { get { return cols; } }
    public string Name { get { return name; } set { name = value; } }

    public Matrix(int rows, int cols) {
      if (rows <= 0 || cols <= 0)
        throw new ArgumentException();
      this.rows = rows;
      this.cols = cols;
      this.data = new float[rows * cols];
      for (int i = 0; i < rows * cols; ++i)
        this.data[i] = 0;
    }

    public Matrix(int rows, int cols, string name) : this(rows, cols) {
      this.name = name;
    }

    public Matrix(int rows, int cols, float[] data) {
      if (rows <= 0 || cols <= 0)
        throw new ArgumentException();
      this.rows = rows;
      this.cols = cols;
      this.data = new float[rows * cols];
      for (int i = 0; i < rows * cols; ++i)
        this.data[i] = data[i];
    }

    public Matrix(int rows, int cols, float[] data, string name) : this(rows,
        cols, data) {
      this.name = name;
    }
```

```
public Matrix(int rows, int cols, bool identity) {
  if (rows <= 0 || cols <= 0)
    throw new ArgumentException();
  this.rows = rows;
  this.cols = cols;
  this.data = new float[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;
    }
  }
}

public Matrix(int rows, int cols, bool identity, string name) : this(rows,
    cols, identity) {
  this.name = name;
}



public Matrix(Matrix source) {
  this.rows = source.rows;
  this.cols = source.cols;
  this.data = new float[source.rows * source.cols];
  this.name = source.name;
  for (int i = 0; i < source.rows * source.cols; ++i)
    this.data[i] = source.data[i];
}

public Matrix copy() {
  return new Matrix(rows, cols, data, name);
}

public float this[int r, int c] {
  get {
    if (r < 0 || c < 0 || r >= rows || c >= cols)
      throw new ArgumentException("Index out of range");
    return data[r * cols + c];
  }
  set {
    if (r < 0 || c < 0 || r >= rows || c >= cols)
      throw new ArgumentException("Index out of range");
    data[r * cols + c] = value;
  }
}
```

```csharp
public Matrix getColumn(int c) {
  if (c < 0 || c >= cols)
    throw new ArgumentException();
  Matrix column = new Matrix(rows, 1);
  for (int r = 0; r < rows; ++r)
    column[r, 0] = this[r, c];
  return column;
}

public void setColumn(int c, Matrix column) {
  if (c < 0 || c >= cols || column.cols != 1 || column.rows != rows)
    throw new ArgumentException();
  for (int r = 0; r < rows; ++r)
    this[r, c] = column[r, 0];
}

public void print() {
  Console.Write("=== {0} x {1}", rows, cols);
  if (name != null && name != "")
    Console.Write(" ({0})", name);
  Console.WriteLine(" ===");
  for (int r = 0; r < rows; ++r) {
    Console.Write(data[r * cols]);
    for (int c = 1; c < cols; ++c)
      Console.Write(" " + data[r * cols + c]);
    Console.WriteLine();
  }
  Console.WriteLine("======");
}

public static Matrix operator +(Matrix self, Matrix other) {
  if (self.rows != other.rows && self.cols != other.cols)
    throw new ArgumentException("Matrix sizes are not equal");
  Matrix result = new Matrix(self.rows, self.cols);
  for (int r = 0; r < self.rows; ++r) {
    for (int c = 0; c < self.cols; ++c) {
      result[r, c] = self[r, c] + other[r, c];
    }
  }
  return result;
}

public static Matrix operator -(Matrix self, Matrix other) {
  if (self.rows != other.rows && self.cols != other.cols)
    throw new ArgumentException("Matrix sizes are not equal");
  Matrix result = new Matrix(self.rows, self.cols);
```

```csharp
    for (int r = 0; r < self.rows; ++r) {
      for (int c = 0; c < self.cols; ++c) {
        result[r, c] = self[r, c] - other[r, c];
      }
    }
    return result;
  }

  public static Matrix operator *(Matrix self, Matrix other) {
    if (self.cols != other.rows)
      throw new ArgumentException("Matrices are not compatible");
    Matrix result = new Matrix(self.rows, other.cols);
    for (int i = 0; i < self.rows; ++i) {
      for (int j = 0; j < other.cols; ++j) {
        float sum = 0;
        for (int k = 0; k < self.cols; ++k) {
          sum += self[i, k] * other[k, j];
        }
        if (Math.Abs(sum) < 0.00001f)
          sum = 0.0f;
        result[i, j] = sum;
      }
    }
    return result;
  }

  public static Matrix operator *(Matrix self, float factor) {
    Matrix result = new Matrix(self.rows, self.cols);
    for (int r = 0; r < self.rows; ++r) {
      for (int c = 0; c < self.cols; ++c) {
        result[r, c] = self[r, c] * factor;
      }
    }
    return result;
  }

  public static Matrix operator *(float factor, Matrix self) {
    return self * factor;
  }

  public Matrix transpose() {
    Matrix result = new Matrix(cols, rows);
    for (int r = 0; r < rows; ++r) {
      for (int c = 0; c < cols; ++c) {
        result[c, r] = this[r, c];
      }
    }
```

```csharp
    return result;
  }

  public void multiplyRow(int row, float factor) {
    if (factor == 1) return;
    for (int j = 0; j < cols; ++j) {
      this[row, j] *= factor;
    }
  }

  public void addRows(int sourceRow, int destinationRow, float factor) {
    if (factor == 0) return;
    for (int j = 0; j < cols; ++j) {
      this[destinationRow, j] += this[sourceRow, j] * factor;
    }
  }

  public void swapRows(int firstRow, int secondRow) {
    if (firstRow == secondRow) return;
    for (int j = 0; j < cols; ++j) {
      float tmp = this[firstRow, j];
      this[firstRow, j] = this[secondRow, j];
      this[secondRow, j] = tmp;
    }
  }

  public Matrix invert() {
    if (rows != cols) throw new ArgumentException("Non-square matrices
      cannot be inverted");
    Matrix self = copy();
    Matrix inverse = new Matrix(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 new ArgumentException("Matrix is singular");
      // Swap with current row; now the current row has nonzero in this
        column
      self.swapRows(rc, searchRow);
      inverse.swapRows(rc, searchRow);
      float factor = 1 / self[rc, rc];
      self.multiplyRow(rc, factor);
      inverse.multiplyRow(rc, factor);
      for (int r = 0; r < rows; ++r) {
        if (r == rc) continue;
```

```
        float innerFactor = -self[r, rc];
        self.addRows(rc, r, innerFactor);
        inverse.addRows(rc, r, innerFactor);
      }
    }
    return inverse;
  }
 }
}
```

Program.cs

Driver code.

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
using System.Globalization;
using System.Threading;

namespace CS_asynplex {
  class PrintControl {
    public bool B = false;
    public bool N = false;
    public bool cBasic = false;
    public bool cNonbasic = false;
    public bool b = false;
    public bool piTranspose = false;
    public bool reducedCosts = false;
    public bool stepDirection = false;
    public bool stepLength = false;
    public bool pivotDecisions = false;
    public bool pivotResults = false;
    public bool variableLists = false;
    public bool iterationCount = false;
    public bool iterationSeparator = false;
    public bool objectiveValue = false;
    public bool etaFileSize = false;
    public bool inverseDiscrepancy = false;
  }

  class Program {
    const float EPS = 0.0000005f;

    private static bool equalsEpsilon(float a, float b) {
```

```csharp
    return Math.Abs(a - b) < EPS;
}

private static object finishedLock = new object();
private static bool _finished = false;
public static bool Finished {
  get {
    lock (finishedLock) {
      return _finished;
    }
  }
  set {
    lock (finishedLock) {
      _finished = value;
    }
  }
}

static void Main(string[] args) {
  StreamReader sr = new StreamReader(args[0]);
  string[] items = sr.ReadLine().Split(' ');
  //Random random = new Random();

  int m = int.Parse(items[0]) - 1;
  int n = int.Parse(items[1]) - 1;
  Matrix N = new Matrix(m, n, "N");
  Matrix B = new Matrix(m, m, true, "B");
  List<Matrix> etaFile = new List<Matrix>();
  Matrix cNonbasic = new Matrix(n, 1, "cNonbasic");
  Matrix cBasic = new Matrix(m, 1, "cBasic");
  Matrix basicValues = new Matrix(m, 1, "basicValues"); // Starts out as
      the b vector; will always equal B^{-1} * b

  items = sr.ReadLine().Split(' ');
  for (int c = 0; c < n; ++c) {
    cNonbasic[c, 0] = float.Parse(items[c], CultureInfo.InvariantCulture.
        NumberFormat);
  }

  for (int r = 0; r < m; ++r) {
    items = sr.ReadLine().Split(' ');
    for (int c = 0; c < n; ++c) {
      N[r, c] = float.Parse(items[c], CultureInfo.InvariantCulture.
          NumberFormat);
    }
    basicValues[r, 0] = float.Parse(items[n], CultureInfo.InvariantCulture
        .NumberFormat);
```

```csharp
  }
  sr.Close();

  int[] nonbasicVars = new int[n];
  int[] basicVars = new int[m];
  for (int i = 1; i <= n; ++i)
    nonbasicVars[i - 1] = i;
  for (int i = n + 1; i <= n + m; ++i)
    basicVars[i - n - 1] = i;

  bool useAsynplex = true;
  if (useAsynplex) {
    const int numIterationProcesses = 4;
    InversionProcess invertProcessor = new InversionProcess(N,
        numIterationProcesses);
    BasisChangeProcess basisChangeManager = new BasisChangeProcess();
    ColumnSelectionProcess columnSelectionManager = new
        ColumnSelectionProcess(m, n);
    Thread t;
    t = new Thread(invertProcessor.run);
    t.Name = "Invert processor";
    t.Start();
    t = new Thread(basisChangeManager.run);
    t.Name = "Basis change manager";
    t.Start();
    t = new Thread(columnSelectionManager.run);
    t.Name = "Column selection manager";
    t.Start();
    for (int i = 0; i < numIterationProcesses; ++i) {
      IterationProcess iter = new IterationProcess(i,
          numIterationProcesses, m, n, N, basicValues, cNonbasic);
      t = new Thread(iter.run);
      t.Name = "Iteration process " + i;
      t.Start();
    }
    return;
  }

  Matrix originalBInverse = new Matrix(B.Rows, B.Cols, true, "orig B^I");
  PrintControl printControl = new PrintControl() { iterationCount = true,
      objectiveValue = true, pivotResults = true};
  int iterationCount = 1;
  int reinversionPeriod = 5;
  while (true) {
    if (printControl.B)      B.print();
    if (printControl.N)      N.print();
    if (printControl.cBasic)  cBasic.print();
```

```csharp
if (printControl.cNonbasic) cNonbasic.print();
if (printControl.b)      basicValues.transpose().print();

Matrix bInverse;
while (true) {
  try {
    if ((iterationCount - 1) % reinversionPeriod == 0) {
      originalBInverse = B.invert();
      etaFile.Clear();
    }
    if (printControl.etaFileSize) Console.WriteLine("{0} eta matrices"
        , etaFile.Count);
    bInverse = originalBInverse;
    foreach (Matrix eta in etaFile) {
      bInverse = eta * bInverse;
    }
    if (etaFile.Count > 0 && printControl.inverseDiscrepancy) {
      float max = 0;
      Matrix discrepancy = bInverse - B.invert();
      for (int r = 0; r < discrepancy.Rows; ++r) {
        for (int c = 0; c < discrepancy.Cols; ++c) {
          float absVal = Math.Abs(discrepancy[r, c]);
          if (absVal > max)
            max = absVal;
        }
      }
      Console.WriteLine("Largest discrepancy: {0}", max);
    }
    break;
  }
  catch (ArgumentException) {
    throw;
    //Console.WriteLine("reinverting");
    //int i = random.Next(0, m), j;
    //do {
    //    j = random.Next(0, m);
    //} while (j == i);
    //Matrix col = B.getColumn(i);
    //B.setColumn(i, B.getColumn(j));
    //B.setColumn(j, col);
    //continue;
  }
}

//BTRAN
Matrix piTranspose = cBasic.transpose() * bInverse; // \pi^T
piTranspose.Name = "pi^T";
```

```csharp
    if (printControl.piTranspose) piTranspose.print();

    //PRICE
    Matrix reducedCosts = cNonbasic.transpose() - piTranspose * N; // \hat
        {c}_N^T
    reducedCosts.Name = "reduced costs";
    if (printControl.reducedCosts) reducedCosts.print();

    if (printControl.objectiveValue) {
      Matrix objectiveValue = cBasic.transpose() * basicValues;
      Console.WriteLine("Objective value: {0}", objectiveValue[0, 0]);
    }

    //CHUZC
    int enteringCol = -1; // q
    for (int c = 0; c < reducedCosts.Cols; ++c) {
      if (reducedCosts[0, c] > 0) {
        Console.Write(c + " ");
      }
      if (reducedCosts[0, c] > 0
        && (enteringCol == -1
          || (reducedCosts[0, c] > reducedCosts[0, enteringCol] && !
              equalsEpsilon(reducedCosts[0, c], reducedCosts[0,
              enteringCol]))
          || (equalsEpsilon(reducedCosts[0, c], reducedCosts[0,
              enteringCol]) && nonbasicVars[c] < nonbasicVars[enteringCol
              ]))) {
        if (printControl.pivotDecisions) Console.WriteLine("Choosing {0}
            over {1} to enter; reduced cost is {2}", c + 1, enteringCol +
            1, reducedCosts[0, c]);
        enteringCol = c;
      }
    }
    Console.WriteLine();
    if (enteringCol == -1) {
      Console.WriteLine("Optimal");
      break;
    }
    int enteringVar = nonbasicVars[enteringCol];
    if (printControl.pivotResults) Console.WriteLine("Entering variable: x
        {1} (column {0})", enteringCol, enteringVar);

    //FTRAN
    Matrix stepDirection = bInverse * N.getColumn(enteringCol); // \hat{a}
        _q
    stepDirection.Name = "step direction";
    if (printControl.stepDirection) stepDirection.print();
```

```csharp
//CHUZR
int leavingRow = -1; // p
float largestRatio = 0; // step length (\alpha)
for (int r = 0; r < basicValues.Rows; ++r) {
  float ratio;
  if (stepDirection[r, 0] == 0 && basicValues[r, 0] == 0)
    ratio = 0;
  else
    ratio = stepDirection[r, 0] / basicValues[r, 0];
  if (ratio > 0 && (leavingRow == -1 || ratio > largestRatio || (ratio
      == largestRatio && basicVars[r] < basicVars[leavingRow]))) {
    if (printControl.pivotDecisions) Console.WriteLine("Choosing {0}
        over {1} to leave; ratio is {2}", r + 1, leavingRow + 1, ratio
        );
    leavingRow = r;
    largestRatio = ratio;
  }
}
if (leavingRow == -1) {
  Console.WriteLine("Unbounded");
  break;
}
float stepLength = 1 / largestRatio;
int leavingVar = basicVars[leavingRow];
if (printControl.pivotResults) Console.WriteLine("Leaving variable: x
   {1} (row {0})", leavingRow, leavingVar);
if (printControl.stepLength) Console.WriteLine("Step length: {0}",
   stepLength);

//UPRHS
basicValues -= stepLength * stepDirection;
basicValues[leavingRow, 0] = stepLength;
basicValues.Name = "b";

//UPDATE (should normally also calculate/update inverse here)
Matrix tempM = N.getColumn(enteringCol);
N.setColumn(enteringCol, B.getColumn(leavingRow));
B.setColumn(leavingRow, tempM);

float dxi = stepDirection[leavingRow, 0];
if (dxi == 0) throw new Exception();
Matrix etaMatrix = new Matrix(B.Rows, B.Cols, true, "Eta" + etaFile.
   Count);
for (int r = 0; r < etaMatrix.Rows; ++r) {
  if (r == leavingRow)
    etaMatrix[r, leavingRow] = 1 / dxi;
```

```
    else
      etaMatrix[r, leavingRow] = -stepDirection[r, 0] / dxi;
  }
  etaFile.Add(etaMatrix);

  float tempF = cNonbasic[enteringCol, 0];
  cNonbasic[enteringCol, 0] = cBasic[leavingRow, 0];
  cBasic[leavingRow, 0] = tempF;

  int tempI = nonbasicVars[enteringCol];
  nonbasicVars[enteringCol] = basicVars[leavingRow];
  basicVars[leavingRow] = tempI;

  if (printControl.variableLists) {
    Console.Write("Basic:    ");
    for (int i = 0; i < m; ++i)
      Console.Write(basicVars[i] + " ");
    Console.Write("\nNonbasic: ");
    for (int i = 0; i < n; ++i)
      Console.Write(nonbasicVars[i] + " ");
    Console.WriteLine();
  }
  if (printControl.iterationCount) Console.WriteLine("\nCompleted
      iteration {0}", iterationCount++);
  if (printControl.iterationSeparator) Console.WriteLine("\n\n");
      }
    Console.ReadLine();
    }
  }
}
```

A.4   ASYNPLEX for x86, based on Vanderbei

(TODO: Describe each file: purpose, and if we have changed it)

A.5   ASYNPLEX for Cell/B.E., based on Vanderbei

(TODO: Describe each file: purpose, and if we have changed it) The complete code base, with Eclipse project files and all the Vanderbei files that we have not modified, can be found in the electronic attachment in the thesis submission system DAIM.

A.6   (RP) Utilities

We could not find any available parsers for the MPS or CPLEX file formats, so we had to write our own. Other people may find them useful, so we include them here. Common languages of choice for writing small text manipulation programs are Python and Perl; we selected the former since we are more familiar with it.

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

(RP) `mps.py` — MPS file format parser

This parser was written in the early stages of the project, when our standard simplex solver would simply expect a full tableau as input. The parser first outputs a line containing m (the number of rows) and n (the number of columns), followed by m lines containing n numbers each. The first row contains the objective function coefficients, and the leftmost column contains the right hand sides from the constraints. The tableau body contains the negatives of the original coefficients, as per our discussion in Section 2.1.2. Equality constraints are split into two less-than constraints.

We later rewrote the parser to C++ (`mps.cpp` as listed above), so that it could be an integrated part of our solver.

Note that both this parser and the C++ port are fairly simplistic, and they do *not* handle the BOUNDS or RANGES sections. As such, the number of `netlib` sets on which they (and thereby our entire standard simplex solver) can be used is reduced to 54 (from a total of 98); see the `netlib` README file for information on which sets contain which sections. Also, note that since the MPS format does not specify the direction of optimisation, and the `netlib` default seems to be minimisation, the CPLEX parser will negate the objective function for all maximisation data sets.

```python
#!/usr/bin/python
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:
```

```python
    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
      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
        i += 1
      for j in xrange(len(tableau)):
        tableau[j] = [0] * (len(columnLabels) + 1)
      for row in rows.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
        i += 1

for row in rows.values():
  tab = tableau[row.index]
  if row.type == "G":
    for i in xrange(len(tab)):
      tab[i] = -tab[i]
  elif row.type == "E":
    tableau.append([-x for x in tab])
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 len(tableau), len(tableau[0])
for tab in tableau:
  for cell in tab:
    print cell,
  print
```

(RP) `cplex.py` — ILOG CPLEX file format parser

This parser was written in order to convert some sample data sets we received from Miriam AS to the MPS format. A restriction of the MPS format is that the row and column names are limited in length. Therefore, our parser will convert any name that is longer than eight characters to a string that is formed by appending a sequence number (starting at zero) to the string v (a very arbitrary choice). For instance, the fourth name that is found to be too long will be converted to v3. Further occurrences of the same name will of course be replaced by the same string. The parser does *not*,

however, check for name collisions with variables who actually have that name.

Note that while the CPLEX format allows constraints to be split over multiple lines, this parser not handle that, so files containing split constraints must be modified by joining such constraints into one line.

```python
#!/usr/bin/python
from sys import stdin, stderr

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):
    self.variable = variable

variableCodeNames = {}

def truncate(name):
  global variableCodeNames
  if len(name) <= 8:
    return name
  else:
    if variableCodeNames.has_key(name):
      return variableCodeNames[name]
    else:
      codeName = "v" + str(len(variableCodeNames))
      variableCodeNames[name] = codeName
      return codeName

def printCodeNames():
```

```python
  global variableCodeNames
  if len(variableCodeNames) > 0:
    stderr.write("Some variable names have been changed:\n")
    stderr.write("New\tOriginal\n")
  for name in variableCodeNames:
    stderr.write(variableCodeNames[name] + "\t" + name + "\n")

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 printMPS(self):
    print "NAME          UNKNOWN"
    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)
```

```python
        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 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:
        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:
        bound = Bound(tokens[2])
        bound.lower = float(tokens[0])
        bound.upper = float(tokens[4])
```

```python
            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()
lp.printMPS()
printCodeNames()
```

(RP) Data sets

The full `netlib` collection is freely available online (see Section 2.3.1), so we only produce one small set here in order to show an example of a file in the MPS format. The answers to all `netlib` sets can be found in the `netlib` README file. We also received four small sets (extracted from a larger data set) from Miriam AS.

B.1 (RP) Sample `netlib` data set — `afiro.mps`

This is the `afiro` set, in MPS format. It contains 32 variables and 27 constraints, and 88 nonzero coefficients. The answer, according to the README file, is $-4.6475314286 \cdot 10^2$.

```
**************************
* SET UP THE INITIAL DATA *
**************************
NAME          AFIRO
*    Problem:
*    ********
*    An LP, contributed by Michael Saunders.
*    Source:
*    The NETLIB collection of test problems.
*    SIF input: (already in MPS format)
*    classification LLR2-AN-32-27
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  COST
COLUMNS
    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   (RP) Data sets provided by Miriam AS

These sets are in the ILOG CPLEX format. Miriam AS also provided us with the answers found by CPLEX.

dp_0.lp

Answer: $4.20001 \cdot 10^2$.

```
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

Answer: $1.001 \cdot 10^0$.

```
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
```

dp_170.lp

Answer: $1.00001 \cdot 10^2$.

```
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  = 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 >= 99.999
      Comp557 Free
      zMax558 >= -0.001
      id105 = 0
 0 <= RgCapE50 <= 200
 0 <= RgCapE51 <= 200
 0 <= RgCapE52 <= 200
 0 <= RgCapS59 <= 120
 0 <= RgCapS60 <= 200
 0 <= RgCapI89 <= 900000000
 0 <= RgCapD53 <= 200
 0 <= RgCapD54 <= 200
End
```

gas.lp

Answer: $2.0009824199873 \cdot 10^1$.

```
Maximize
 obj: zMax3873 + 0.000001 tieBreak0 + id214
Subject To
 R2_3362:              - v3362 + v3362_3371  = 0
 R10_3362_3371:        - v3362_3371 + x3856_3371  = 0
 R2_3364:              - v3364 + v3364_3371  = 0
 R6_3364_3371:         x3856_3371 - v3364_3371  = 0
 R18_3856:             - x3856 + 10 y3856 >= 0
 R21_3856:             _inletPressure3364 <= 10000
 R25_3364_0:           - 1.42072309829124 _inletPressure3364 +
    _outletPressure3364 <= 8441.5873846062
```

```
R25_3364_1:             - 1.42916309321641 _inletPressure3364 +
   _outletPressure3364 <= 8324.69345489257
R25_3364_2:             - 1.43813237097197 _inletPressure3364 +
   _outletPressure3364 <= 8205.40206074365
R25_3364_3:             - 1.4476886789981 _inletPressure3364 +
   _outletPressure3364 <= 8083.55913341047
R25_3364_4:             - 1.45789899041056 _inletPressure3364 +
   _outletPressure3364 <= 7958.99333417841
R25_3364_5:             - 1.46884149019766 _inletPressure3364 +
   _outletPressure3364 <= 7831.5132116587
R25_3364_6:             - 1.48060811048304 _inletPressure3364 +
   _outletPressure3364 <= 7700.90372649103
R25_3364_7:             - 1.49330780379467 _inletPressure3364 +
   _outletPressure3364 <= 7566.9219620533
R11_3364_3371:          v3364_3371 - x3858_3371  = 0
R33_3364_3355:          - F3364_3355 + F3856_3355  = 0
R35_3364_3355:          QDown3364_3355 - QUp3364_3355  = 0
R33_3364_3356:          - F3364_3356 + F3856_3356  = 0
R35_3364_3356:          QDown3364_3356 - QUp3364_3356  = 0
R33_3364_3357:          - F3364_3357 + F3856_3357  = 0
R35_3364_3357:          QDown3364_3357 - QUp3364_3357  = 0
R33_3364_3358:          - F3364_3358 + F3856_3358  = 0
R35_3364_3358:          QDown3364_3358 - QUp3364_3358  = 0
R33_3364_3359:          - F3364_3359 + F3856_3359  = 0
R35_3364_3359:          QDown3364_3359 - QUp3364_3359  = 0
R33_3364_3360:          - F3364_3360 + F3856_3360  = 0
R35_3364_3360:          QDown3364_3360 - QUp3364_3360  = 0
R33_3364_3361:          - F3364_3361 + F3856_3361  = 0
R35_3364_3361:          QDown3364_3361 - QUp3364_3361  = 0
R2_3363:                - v3363 + v3363_3371  = 0
R6_3363_3371:           x3858_3371 - v3363_3371  = 0
R14_3363:               totalDelivery3363_3371 <= 10
R19_3858:               - x3858 + 10 y3858 >= 0
R23_3858:               _outletPressure3364 >= 14400
R33_3363_3355:          - F3363_3355 + F3858_3355  = 0
R33_3363_3356:          - F3363_3356 + F3858_3356  = 0
R33_3363_3357:          - F3363_3357 + F3858_3357  = 0
R33_3363_3358:          - F3363_3358 + F3858_3358  = 0
R33_3363_3359:          - F3363_3359 + F3858_3359  = 0
R33_3363_3360:          - F3363_3360 + F3858_3360  = 0
R33_3363_3361:          - F3363_3361 + F3858_3361  = 0
R12_3363_3371:          - v3363_3371 + d3363_3371  = 0
R13_3363_3371:          - totalDelivery3363_3371 + d3363_3371  = 0
AggR5_3856:             x3856_3371 - x3856  = 0
R36_3856_3355:          F3856_3355  = 0
R36_3856_3356:          F3856_3356  = 0
R36_3856_3357:          F3856_3357  = 0
```

```
R36_3856_3358:         F3856_3358  = 0
R36_3856_3359:         F3856_3359  = 0
R36_3856_3360:         F3856_3360  = 0
R36_3856_3361:         F3856_3361  = 0
AggR5_3858:            x3858_3371 - x3858  = 0
Max3873:               - d3363_3371 + zMax3873  = 0
MaxTieBreak3873:       _inletPressure3364 - _outletPressure3364 + tieBreak0 =
    0
Bilin3859F3364_33550: F3364_3355 - 10 QUp3364_3355 + slack3859F3364_33550  =
    0
Bilin3860F3364_33560: F3364_3356 - 10 QUp3364_3356 + slack3860F3364_33560  =
    0
Bilin3861F3364_33570: F3364_3357 - 10 QUp3364_3357 + slack3861F3364_33570  =
    0
Bilin3862F3364_33580: F3364_3358 - 10 QUp3364_3358 + slack3862F3364_33580  =
    0
Bilin3863F3364_33590: F3364_3359 - 10 QUp3364_3359 + slack3863F3364_33590  =
    0
Bilin3864F3364_33600: F3364_3360 - 10 QUp3364_3360 + slack3864F3364_33600  =
    0
Bilin3865F3364_33610: F3364_3361 - 10 QUp3364_3361 + slack3865F3364_33610  =
    0
Bilin3866F3858_33550: - 10 QDown3364_3355 + F3858_3355 + slack3866F3858_33550
     = 0
Bilin3867F3858_33560: - 10 QDown3364_3356 + F3858_3356 + slack3867F3858_33560
     = 0
Bilin3868F3858_33570: - 10 QDown3364_3357 + F3858_3357 + slack3868F3858_33570
     = 0
Bilin3869F3858_33580: - 10 QDown3364_3358 + F3858_3358 + slack3869F3858_33580
     = 0
Bilin3870F3858_33590: - 10 QDown3364_3359 + F3858_3359 + slack3870F3858_33590
     = 0
Bilin3871F3858_33600: - 10 QDown3364_3360 + F3858_3360 + slack3871F3858_33600
     = 0
Bilin3872F3858_33610: - 10 QDown3364_3361 + F3858_3361 + slack3872F3858_33610
     = 0
Bounds
 0 <= v3362 <= 10
 0 <= v3362_3371 <= 20
 0 <= v3364 <= 10
     y3856 = 1
 0 <= _inletPressure3364 <= 10000
     _outletPressure3364 >= 14400
     F3364_3355 Free
     F3856_3355 Free
     QUp3364_3355 = 0
     F3364_3356 Free
```

```
      F3856_3356 Free
      QUp3364_3356 = 0
      F3364_3357 Free
      F3856_3357 Free
      QUp3364_3357 = 0
      F3364_3358 Free
      F3856_3358 Free
      QUp3364_3358 = 0
      F3364_3359 Free
      F3856_3359 Free
      QUp3364_3359 = 0
      F3364_3360 Free
      F3856_3360 Free
      QUp3364_3360 = 0
      F3364_3361 Free
      F3856_3361 Free
      QUp3364_3361 = 0
0 <= v3363 <= 20
0 <= v3363_3371 <= 20
0 <= totalDelivery3363_3371 <= 10
      y3858 = 1
      F3363_3355 Free
      F3858_3355 Free
      F3363_3356 Free
      F3858_3356 Free
      F3363_3357 Free
      F3858_3357 Free
      F3363_3358 Free
      F3858_3358 Free
      F3363_3359 Free
      F3858_3359 Free
      F3363_3360 Free
      F3858_3360 Free
      F3363_3361 Free
      F3858_3361 Free
      zMax3873 Free
      tieBreak0 Free
      slack3859F3364_33550 = 0
      slack3860F3364_33560 = 0
      slack3861F3364_33570 = 0
      slack3862F3364_33580 = 0
      slack3863F3364_33590 = 0
      slack3864F3364_33600 = 0
      slack3865F3364_33610 = 0
      slack3866F3858_33550 = 0
      slack3867F3858_33560 = 0
      slack3868F3858_33570 = 0
```

```
      slack3869F3858_33580 = 0
      slack3870F3858_33590 = 0
      slack3871F3858_33600 = 0
      slack3872F3858_33610 = 0
      id214 = 0
End
```