11-15-2006

Cellular Oxidative Efficiency: A New Approach to Calculating Theoretical P/O Ratios

Douglas Walled

Cellular Oxidative Efficiency:
A New Approach to Calculating Theoretical P/O Ratios

Douglas Walled

Yale University School of Medicine

2006

Cellular Oxidative Efficiency: A New Approach to Calculating Theoretical P/O Ratios

A Thesis Submitted to the
Yale University School of Medicine
in Partial Fulfillment of the Requirements for the
Degree of Doctor of Medicine

by

Douglas Walled

2006

CELLULAR OXIDATIVE EFFICIENCY: A NEW APPROACH TO CALCULATING THEORETICAL P/O RATIOS. Douglas G. Walled (Sponsored by Paul K. Maciejewski) Magnetic Resonance Research Center, Department of Psychiatry, Yale University, School of Medicine, New Haven, CT.

For decades the oxidative efficiency of cellular metabolism has been under investigation. After numerous reports of varied stoichiometric measurements, consensus in the literature has begun moving toward two currently accepted theoretical P/O ratios (the number of adenosine triphosphate (ATP) molecules formed for every oxygen atom consumed): 2.5 for NADH-linked substrates and 1.5 for $FADH_2$-linked substrates. It is shown here, however, that the currently accepted theoretical values are inappropriately calculated underestimates, and that P/O ratios of real biochemical systems are variable.

The complete oxidative metabolism of glucose, beta-hydroxybutyrate, malate, pyruvate, and succinate, utilizing three different electron shuttles (or exclusive mitochondrial metabolism) and two different values of the H^+/ATP ratio (4 and 13/3) is examined using a new method of analysis. Calculations are made within the rigid mathematical framework of linear algebra, relying on the Law of Conservation of Matter as a first principle.

Calculated P/O values from systems modeled after cell-free mitochondrial extracts ranged from 2.711 to 3.183, or 3.000 to 3.500 depending on H^+/ATP ratios of 13/3 or 4/1, respectively. These estimates are within the range of measured values (1.07 - 3.73) but are higher than the commonly accepted theoretical values of ~2.5 and ~1.5 for NADH and $FADH_2$-linked substrates, respectively. A new view of the P/O ratio as variable, based on

specific details of molecular physiology, is offered as a potentially useful means for understanding variation in measured values of the P/O ratio reported in the literature.

Acknowledgements

Thanks to the Department of Student Research at Yale University School of Medicine for providing financial support.

Thanks to Paul K. Maciejewski of the Department of Psychiatry at Yale School of Medicine for the guidance and education that made this research possible.

TABLE OF CONTENTS

Introduction

For decades the stoichiometry of cellular oxidative phosphorylation has been under investigation as a means of measuring metabolic efficiency. Since the realization that phosphorylation reactions were coupled with the oxidation of organic compounds in the earlier part of last century (1, 2) biochemists have made countless attempts to quantify the stoichiometric ratio for this process. The characteristic measure originally implemented and now entrenched by tradition is the P/O ratio: the number of adenosine triphosphate (ATP) molecules formed for every oxygen atom consumed. Recent literature and textbooks suggest that there are two set mechanistic values of the P/O ratio (to be described shortly), and that the process of theoretically deriving these ratios is well understood and supported by experimental measurements. It will be shown, however, that the currently accepted theoretical values are inappropriately calculated underestimates. In addition, the P/O ratio of real biochemical systems should be considered variable, instead of being forced into one of two mechanistic values. The potential ratios span an essentially continuous range depending on a number of factors, and the true calculation of theoretical P/O ratios is more complex than has been considered to date. After describing the historical issues of measuring and calculating the P/O ratio, a new methodology will be outlined and applied. This method will supply more reliable estimates of theoretical P/O ratios for several substrates, and offer insight into conflicting measurements seen in the literature.

Measuring P/O, An Historic Problem

The P/O ratio (using pyruvate as a substrate) was initially measured by Kalckar and Belitzer to have a value of 2 (1,3), and shortly re-determined with confidence to be 3 in the oft cited work by Ochoa in 1943 (4). It was thought that this single value represented the efficiency of aerobic respiration. By extension, considering the chemical equation for the complete oxidation of glucose should give the theoretical ATP yield for metabolizing that substrate.

$$1\ C_6H_{12}O_6\ +\ 6\ O_2\ \rightarrow\ 6\ CO_2\ +\ 6\ H_2O$$

If 12 atoms of oxygen (from 6 molecules of O_2) are consumed by machinery that produces three high energy phosphate bonds per oxygen atom consumed (P/O = 3), then 36 molecules of ATP must be produced when glucose is fully oxidized. This first suggestion still survives as dogma in some biochemistry textbooks today (5). Predicting ATP yields will be considered later in the discussion section, however the rest of the Introduction will focus specifically on the problem of the P/O ratio.

Conflict over the value of the P/O ratio has existed since its inception. Dissenters such as Lehninger and Bartley sided with Kalckar and Belitzer, reporting values closer to 2, and criticizing Ochoa's value of 3 (which was reported as a corrected value in his original

publication) as having been “arrived at by making corrections of questionable validity for dephosphorylation." (6, 7) Still others backed Ochoa, tauting values of 3 (8). Although no set P/O ratio was officially settled upon in the 40s and 50s, people did agree that the mechanistic ratio *must* be an integer (most reported measurements were not integers, but were rounded as a result of error correction in the discussion).

The Chemiosmotic Hypothesis, a paradigm shift

The introduction of the chemiosmotic hypothesis (9) removed the restriction that the P/O ratio must be an integer. Chemiosmotic Theory describes oxidative metabolism as two independent but linked processes. The first process is the translocation of protons from the mitochondrial matrix to the cytosol, coupled to the oxioreductive reactions of the electron transport chain (ETC). The second process is the opposite—translocation of protons from the cytosol to the matrix, coupled to the phosphorylation reaction that makes ATP (10). Reduced electron carriers produced in earlier steps of energy metabolism (such as NADH and $FADH_2$) funnel their electrons into molecular Oxygen (O_2) through the massive protein complexes of the ETC. As these electrons flow in a steady state system, protons moving into the matrix, driving the ATPase to phosphorylate ATP, balance the protons moving out through the ETC complexes. These complimentary processes result in the maintenance of a constant chemiosmotic gradient. Since these processes are linked through the common medium of protons, it is not necessary that there be a whole number ratio between ATP produced, and oxygen consumed. Instead, exactly balancing proton translocations to maintain a steady state determines the ratio.

P/O Measures Continue to No Avail

After the introduction of the chemiosmotic hypothesis, numerous P/O ratios were experimentally determined using many different methodologies and several different substrates. Though the system studied was almost always a cell-free mitochondrial extract, the organ and species from which the native tissues were harvested varied. Experiments were carried out in a wide variety of prepared media containing (most commonly) one of the following five substrates: glucose, β-hydroxybutyrate, malate, pyruvate, and succinate.

Reported P/O ratios from the literature have appeared in various forms: as ranges, means with standard deviations, or even single values with no mention of error (especially in earlier reports). Since these inconsistent measures cannot be meaningfully compared with statistics, they are reviewed here as ranges from the lowest to the highest possible measure reported within a publication (see Table 1). Over the last sixty years, biochemists have reported P/O ratios for five substrates spanning from 1.07 to 2.2 for succinate and 1.86 to 3.73 for the other four common substrates (see Figure 1). Since the succinate to fumarate oxidation step produces an $FADH_2$ and bypasses the first part of the ETC, many have tried to consider P/O ratios associated with it to be distinct from other substrates that produce mostly NADH in their oxidation. In order to force this result, studies measuring the P/O ratio of succinate oxidation generally include an inhibitor that blocks complex I of the ETC, often lowering the measured P/O ratio (first demonstrated experimentally by Greengard et al, 1959). The ETC, $FADH_2$-linked, and NADH-linked substrates will be discussed in detail shortly.

The variability of measured P/O ratios, reflecting disagreement in the literature, led to many analyses and reviews accounting for how others had miscalculated, over or under corrected, or simply mismeasured the P/O ratio. Researchers began striving to simplify the wealth of findings. Currently, many biochemists believe that the issue is practically settled, and that definitive P/O values exist. The trend is to correct the numerous measurements toward a P/O ratio of 1.5 for succinate- or $FADH_2$-linked substrates, and 2.5 for NADH-linked substrates (11, 12, 13), values that appear in current textbooks (14, 15). Some contemporary studies have continued to question these values however, claiming that 1.5 and 2.5 are underestimates (16, 17, 18), an opinion shared by the author to be justified herein.

Calculating the P/O Ratio, A New Hope

By 1980 the state of disagreement between P/O measurements drove biochemists to begin investigating a means to calculate the P/O ratio. Some groups attempted sophisticated methods of calculating P/O ratios utilizing non-equilibrium thermodynamics (19, 20, 21, 22). However, these calculations rely on measured values of concentrations of metabolic intermediates. Due to the reliance upon measured values, they are not purely theoretical, and are subject to the same experimental errors measured P/O ratios have suffered in the past.

The more commonly accepted way to calculate a theoretic P/O ratio is a simple arithmetic combination of two other ratios given by the chemiosmotic theory. These are the $H^+/2e^-$ ratio (also called H^+/O ratio because one atom of oxygen accepts two electrons as the final reaction of the ETC) and the H^+/ATP ratio. These ratios, their values, and how they are combined to calculate currently accepted P/O ratios are described below.

The $H^+/2e^-$ Ratio

The $H^+/2e^-$ ratio is the number of protons translocated by an ETC complex per two electrons passing through its oxioreductive center. The ETC contains four complexes (I-NADH/Q reductase, II-Succinate/Q reductase, III-cytochrome reductase, and IV-cytochrome oxidase). Since only complexes I, III, and IV translocate protons (thus having an $H^+/2e^-$ ratio), they are often referred to as site 1, site 2, and site 3 of the ETC, respectively. Earlier studies demonstrated an $H^+/2e^-$ ratio of four for at all three sites (23, 24, 25, 26). These studies did not take into consideration the important distinction between scalar protons and vector protons, however. A scalar proton is one that is produced or consumed in the cytosol or matrix without being consumed or produced, respectively, in the opposite compartment. A vector proton (i.e. one that is transported) is consumed in one compartment and produced in the other with a 1:1 ratio.

Other studies used the notion of scalar and vector protons to give a more precise indication of $H^+/2e^-$ ratios at individual ETC complexes. For example, even though Villalobo demonstrated that four protons appear in the cytosol as two electrons flow through site 2 (25), this does not mean that site 2 pumps four protons. In fact, it is now widely accepted that site 2 pumps two protons vectorily, and 2 scalar protons appear in the cytosol when two electrons flow through it (27). Similarly, Antonini et. al. have clearly shown that when two electrons flow through site 3, two scalar protons disappear from the matrix and two protons are vectorily pumped to the cytosol (28), which agreed with previous work by Thelen et al. (29). Site 1 is believed to truly pump four protons (30), though a scalar proton does still

disappear from the matrix (30). The exact mechanisms of all reactions within the ETC complexes (especially proton pumping) are still not completely known, thus currently accepted $H^+/2e^-$ ratios are not absolutely certain. However, it is generally accepted as accurate that site 1 translocates four protons with an additional proton disappearing from the matrix, site 2 translocates two protons with two additional protons appearing in the cytosol, and site 3 translocates two protons with additional protons disappearing from the matrix. See Figure 2a & 2b for a complete summary of the ETC as described.

H^+/ATP Ratio

The maintenance of the electrochemical gradient generated by the electron transport chain drives ATP formation via the inner mitochondrial membrane-associated ATPase. Traditionally, the H^+/ATP ratio (the number of protons translocated from cytosol to matrix per ATP produced) is said to have two components: transport and ATP formation. The transport component is generally accepted as a single proton cotransported into the matrix with an orthophosphate (HP_i^{-1}, or P_i) in an electroneutral exchange (31). The ATP translocase swaps a matrix ATP^{-4} into the cytosol while bringing a cytosolic ADP^{-3} into the matrix in an electrogenic exchange (32) that does not contribute to the H^+/ATP ratio. The second component (how many protons must be transported across the ATPase to drive the production of an ATP) is the subject of some controversy. Mitchell proposed the first value: two protons translocated into the matrix per ATP produced, *including* transport (33). Shortly thereafter, other groups found the ratio to be two *not* including transport, giving an overall H^+/ATP ratio of three (34, 35). The two vs. three debate waged for some time (reviewed in

36). In 1983, a convincing study demonstrated that the overall H^+/ATP ratio should be four including transport (37), the value used most often in P/O ratio calculations. This implies the ATPase requires the translocation of three protons to generate one ATP, a value accepted by many people today. At the time however, some dissenters still insisted on an overall ratio of three (38), and Vink et. al. argued that the H^+/ATP ratio was variable, ranging from 2.15-3.6 (39).

The discrepancy between measurements, as well as advances in technology and methodologies of determining the physical structure of the ATPase, have driven biochemists to a new approach. Many now believe that the H^+/ATP does not have to be measured, but instead is determined by an intrinsic ratio of ATPase subunits. Since the nanomotors comprising the moving core of the ATPase (namely subunits F0 and F1) are mechanically coupled, the ratio of their subunits represents the non-transport portion of the H^+/ATP ratio (40, 41). Since the F1 subunit has three binding sites for ADP, one full revolution of the ATPase core will result in the production of three molecules of ATP. If every F0 subunit c takes up a cytosolic proton for transport as it rotates, after a full revolution the total number of protons transported to make three molecules of ATP is equal to the number of c subunits in F0. Thus, the H^+/ATP ratio should be the number of F0 c subunits divided by three ATPs formed, plus one (for P_i transport). Unfortunately, the number of F0 subunits in mammalian mitochondria has not been definitively measured to the author's knowledge. It is well characterized in several other examples, however (see Table 2). These values lead some to believe the human F0/F1 ratio should be 10/3 (13), resulting in a total H^+/ATP ratio of 13/3 or 4.33 (10/3 H^+ plus 1 H^+ for P_i transport) instead of 4 (3 H^+ plus 1 H^+ for P_i) as cited above.

Current P/O Ratio Calculations: an Oversimplification?

As outlined above, the values for the $H^+/2e^-$ ratio and the H^+/ATP ratio are reasonably agreed upon. Given these ratios, one can argue that the ATP/O ratio (i.e. the P/O ratio) is the sum of the $H^+/2e^-$ ratios of ETC complexes involved in a substrate's complete oxidation, divided by the H^+/ATP ratio. For example, a landmark review calculated the theoretical P/O ratio for the oxidation of NADH and $FADH_2$. By extending the calculation, the theoretical ATP yield for the full oxidation of a single molecule of glucose (reflecting back to Ochoa's original work) is then reported (12). This calculation gives a yield of ~30 ATP per glucose, another value commonly published in textbooks (14, 15). The derivation of the P/O ratio is presented here in depth, but a closer analysis of the ATP yield is reserved for the discussion section.

As mentioned above, a simplification of reviewed $H^+/2e^-$ ratio literature leads the authors of (12) to the conclusion that the H^+/O ratios for sites 1, 2, and 3 are 4:1, 2:1, and 4:1, respectively. Also mentioned earlier, the H^+/ATP ratio is assumed to be 4. Thus, a mechanistic P/O ratio can be derived for each site of the ETC by dividing the two values, thus giving sites 1, 2, and 3 P/O ratios of 1:1, 0.5:1, and 1:1, respectively. Since NADH donates its electrons first at complex I (which subsequently pass through sites 2 and 3), the P/O ratio for a substrate that is oxidized to NADH would be 2.5 (1 + 0.5 + 1). On the other hand, $FADH_2$ donates its electrons at complex II (which then only pass through sites 2 and 3) giving a P/O ratio of 1.5 (0.5 + 1).

Though it may sound complicated, the above calculation is merely an arithmetic combination of several values, and it is grossly oversimplified. A true calculation of a substrate-specific theoretic P/O ratio should be dependent upon several variables and

considerably more involved. First, it should be noted that most substrates will not yield strictly one of two P/O ratios (i.e. one for NADH-linked substrates, and another for $FADH_2$-linked substrates). It should be immediately obvious that substrates are oxidized into a mixture of both NADH- and $FADH_2$- linked intermediates, thus a substrate-specific P/O should never be exactly 2.5 or 1.5. Second, there is an issue of electron shuttling. When a process such as glycolysis produces reduced equivalents (e.g. NADH) in the cytosol, instead of directly transporting NADH (a huge, charged molecule) into the matrix, various shuttles (e.g. the glycerophosphate shuttle or the malate/aspartate shuttle) functionally transport the electrons (each with different costs of transport, and the potential to switch from NADH-linked to $FADH_2$-linked). These shuttles are reviewed below. Third, the existence of a chemiosmotic gradient requires compartmentalization and necessitates accounting for energy consuming transport processes between the cytosol and matrix (which may be different depending on the substrate and/or its intermediates). Fourth, substrate-level phosphorylation can alter the P/O ratio in metabolically similar systems. For example, the two molecules of ATP produced anaerobically in glycolysis mean that glucose will always have a slightly higher P/O ratio than pyruvate when all other conditions are the same. Fifth, allowing the constraint of a steady-state system greatly simplifies the calculation and makes the result more physiologically meaningful (as a respiring mitochondrion in vivo is overall at a steady state). Finally, at steady state, all catabolic reactions associated with the given substrate must be known, and all reaction byproducts must be accounted for simultaneously. Molecular intermediates across all reactions must remain zero. For example, in complete oxidative glucose metabolism, more than 70 molecules participating in over 50 reactions must exactly balance to yield the overall equation:

$$C_6H_{12}O_6 + 6\,O_2 \rightarrow 6\,CO_2 + 6\,H_2O$$

Devising and assuring such a balance is a cumbersome and complex task, particularly one that does not lend itself to mental manipulation.

Electron Shuttles, Altering Metabolic Efficiency

A number of different electron shuttles transport electron equivalents from the cytosol to the matrix. These shuttles utilize different transports and can ultimately change the ETC point of entry of a metabolic intermediate. For example, the glycerophosphate (G3P) shuttle is less efficient because it converts cytosolic NADH equivalents (which enter the ETC at Complex I) to matrix $FADH_2$ equivalents (which enter the ETC at Complex II, and ultimately translocate less protons to be used by the ATPase) (42, 43). The malate-aspartate shuttle (MAS) effectively shuttles NADH equivalents from cytosol to matrix in an irreversible mechanism (44, 45, 46). An NADH equivalent begotten via the MAS, however, is worth slightly less energetically because it requires the transport of one proton with the glutamate/aspartate exchanger See Figure 2a and 2b for mechanistic details.

As another alternative, there is an NADH dehydrogenase (NADH-DH) situated in the mitochondrial membrane facing the cytosol that is linked to Complex I of the ETC. This NADH-DH can directly utilize cytosolic NADH equivalents, but is limited in its distribution. Specifically, the NADH-DH has only been demonstrated to exist in the heart, and has been proven to be absent from the liver (47, 48). See figure 2c for mechanistic details. The utilization of different electron shuttles (of which there may be more than the three

mentioned and examined here), and the proportion in which they are used directly affects the efficiency of oxidative metabolism, thereby altering the P/O ratio.

Summary of Introduction

Many have struggled to define a set stoichiometric relationship between the oxygen consumption and ATP production of energy metabolism for over half of a century, leaving behind a vastly diverse collection of data, theories, and measured ratios in the literature. Discrepant measurements have led to the use of widely accepted H^+/O and H^+/ATP ratios in order to calculate theoretical P/O ratios. These theoretical values are meant to guide a selection process of which measured P/O ratios are the most valid. Unfortunately, these theoretical values have only led to selective criticism of experiments that measured significantly different values. The calculation of the theoretical values themselves have not been scrutinized. Given the potential errors and oversimplification of the calculations outlined above, a more rigorous calculation is required. Such a calculation should be strictly grounded in an indisputable physical law (here, the Law of Conservation of Matter), fully account for all reactions and transport processes, and be purely mathematical. The introduction of (one of several) substrates into a cell-free, respiring mitochondrial system must result in the complete use of ATP (to maintain steady state) generated by the complete oxidative metabolism of those substrates. The result of the calculation should describe all molecular species consumed and produced in the process. Ideally, this will be the balanced oxidation reaction of the substrate **only**.

Purpose

The purpose of this study is to provide accurately computed theoretical P/O ratios for glucose, beta-hydroxybutyrate, malate, pyruvate, and succinate. Results will be based on the fundamental values (e.g. $H^+/2\ e^-$ and H^+/ATP ratios), and reaction mechanisms of oxidative metabolism reviewed from the literature above. They will be calculated within the rigid mathematical framework of linear algebra, relying on the Law of Conservation of Matter as a first principle. Consideration will be given to costs of transport and utilization of various electron shuttles in oxidizing the aforementioned substrates within a steady-state system. It is expected that the P/O ratio will be variable depending on the conditions listed above, and that predictions made in the literature (e.g. theoretical ATP yield from the complete oxidation of

glucose) may be incorrect or misleading because they are based on an oversimplified means of calculation.

Methods

Complex biochemical processes (such as oxidative metabolism) can be represented as a system of linear equations (individual chemical reactions) in terms of matrices and vectors. Though the specific analysis outlined here has not been done before, linear algebraic manipulation of matrices and vectors representing biochemical systems have been described in the past (49, 50). Observe:

$R \cdot \mathbf{v} = \mathbf{b}$ where:
R is an m X n reaction matrix (rows = molecules, columns = reactions)
v is a vector of elements $\{v_n \mid v_n$ is the reaction rate of column n of R$\}$
b is a vector of elements $\{b_m \mid b_m$ is the net production of row m of R$\}$

This matrix equation is justified by the Law of Conservation of Matter. Simply put, elemental pieces of reacting molecules (counts of atomic species and charge) must remain constant across a reaction or, by extension, an arbitrarily large system of reactions. Thus, when set equal to a vector of net consumption/production of participating molecules **b**, the system of equations represented by $R \cdot \mathbf{v} = \mathbf{b}$ may be solved to determine the relative rates of reactions given by **v** (reaction stoichiometry).

To do this, individual biochemical reactions are first encoded into vectors using textbooks and primary literature. These vectors are then combined as the columns of a matrix R to represent the desired system or biochemical process. Using the lower-upper (LU) decomposition algorithm to assist in matrix factorization, the system of equations (represented by the equation $R \cdot \mathbf{v} = \mathbf{b}$) is solved for some given **b** of overall production/consumption of molecules in the system. The solution vector **v** represents the relative rates of reaction within a steady-state system. Comparing any two elements of **v** gives a reaction stoichiometry. When scaled by the coefficients of molecules within balanced chemical equations, the reaction stoichiometries become molecular stoichiometries (e.g. converting the ATPase to Complex IV reaction stoichiometry to an ATP production to Oxygen consumption molecular stoichiometry). A sample calculation is presented here.

Example of Linear Algebraic Calculation

Individual reactions are encoded as vectors whose elements represent the coefficients of the balanced reaction. For example, the first step of glycolysis:

$$\text{ATP} + \text{Glucose} \longrightarrow \text{Glucose-6-Phosphate} + \text{ADP} + \text{H}^+$$

Would translate as the reaction vector:

r1 = {-1, -1, 1, 1, 1}, representing the linear equation:

$-ATP^{4-} - Glucose + Glucose\text{-}6\text{-}Phosphate^{2-} + ADP^{3-} + H^{+} = 0$

Central to the interpretation of this equation is the mass and charge balance required by the Law of Conservation of Matter. A negative coefficient represents consumption in a reaction, and a positive represents production. Close inspection will reveal that the tally of all individual atoms and charges are all perfectly balanced (hence "= 0").

Once these reaction vectors are obtained for an entire system (in this example, the first five reactions of Glycolysis), they are used to construct a matrix. Arbitrarily many 'molecules' (elements) may be added to any reaction vector (as long as the coefficient is 0), and the reaction will still be balanced. Arranging reaction vectors as columns, the rows of the matrix correspond to molecules participating in the system of reactions. For example, the first five reactions of glycolysis:

R_{G5} = {**r1**, **r2**, **r3**, **r4**, **r5**}

R_{G5}	**r1**	**r2**	**r3**	**r4**	**r5**
H+	1	0	1	0	0
ATP	-1	0	-1	0	0
ADP	1	0	1	0	0
Glucose	-1	0	0	0	0
Glucose-6-Phosphate	1	-1	0	0	0
Fructose-6-Phosphate	0	1	-1	0	0
Fructose-1,6-BisPhosphate	0	0	1	-1	0
Glyceraldehyde-3-Phosphate	0	0	0	1	1
Glycerone-3-Phosphate	0	0	0	1	-1

The reaction matrix, R_{G5}, when right multiplied by a vector of relative reaction rates (**v**), gives an overall production/consumption vector (**b**) of individual molecules in the system. Observe:

$\mathbf{v} = \{ v_{r1}, v_{r2}, v_{r3}, v_{r4}, v_{r5} \}$ And: $R_{G5} \cdot \mathbf{v} = \mathbf{b}$ Where:
$\mathbf{b} = \{b_1, b_2, b_3, b_4, b_5, b_6, b_7, b_8, b_9 \}$ And:
b_i is the net production/consumption of the i^{th} molecule (row) in the matrix above

For example:

if $\mathbf{v} = \{ 1, 1, 1, 1, 1 \}$, then $R_{G5} \cdot \mathbf{v} = \mathbf{b}$ Where:
$\mathbf{b} = \{ 2, -2, 2, -1, 0, 0, 0, 2, 0 \}$

This example shows that if (according to **v**) all reactions in the R_{G5} system occur in perfect one-to-one correspondence (1:1 reaction stoichiometry, for all $r_j : r_k$), for each glucose consumed ($b_4 = -1$) with 2 ATP ($b_2 = -2$), there will be 2 protons, 2 ADP, and 2 molecules of Glyceraldehyde-3-Phosphate produced (b_1, b_3, and b_8, respectively). It also tells us Glucose-6-Phosphate, Fructose-6-Phosphate, Fructose-1,6-Bisphosphate, and Glycerone-3-Phosphate are never produced or consumed, regardless of how many cycles of the process occur. In addition, there is a 2:1 molecular stoichiometry of ATPs produced per glucose molecule consumed ($b_2 : b_4$).

Often, the relative rates of reaction (**v**) are unknown, or at least not obvious (as they are in the limited example above). In this case **b** may be specified, and using an LU-decomposition to factor the matrix allows the system (represented by R_{G5}) to be solved for **v**. The specifics of LU-decomposition can be obtained from any Linear Algebra text (e.g. 51, pp. 142-146).

Real biochemical systems (such as oxidative energy metabolism) are almost always "over determined." That is, the matrices that represent them have more rows (molecules) than columns (reactions). In order to facilitate solution, the system of reactions may be represented by splitting the original full matrix R into a square matrix and a rectangular remnant. This introduces some limitations. First, the rows (molecules) included in the square matrix must represent linearly independent equations (see 51, pp. 65-73 for discussion of linearity in this context). Second, the expected overall production/consumption for the molecules included in the square matrix must be known. Third, in order to solve the system, the square matrix must be non-singular (i.e. invertible: see 51, pp. 118-125 for discussion). If an appropriate (that is, non-singular) square matrix R_{g5sqr} and corresponding partial $\mathbf{b_{sqr}}$ vector can be constructed, then **v** can be determined after using the LU decomposition algorithm to factor the square reaction sub-matrix. Once **v** is determined, it can be left-multiplied by the rectangular remnant to give $\mathbf{b_{rmt}}$: the production/consumption of the remaining molecules. Together, $\mathbf{b_{sqr}}$ and $\mathbf{b_{rmt}}$ represent the entire production/consumption (**b**) vector for the original system. Consider the following example:

Assume it is known that consuming glucose in the first five reactions of Glycolysis produces Glyceraldehyde-3-Phosphate, but the stoichiometric ratio of production is not known. Now suppose the relative production and consumption of intermediates are also not known, except the other phosphate bearing intermediates, which are neither produced nor consumed overall. Construct the square matrix:

The Square Matrix, R_{g5sqr}:

R_{g5sqr}	**r1**	**r2**	**r3**	**r4**	**r5**
Glucose-6-Phosphate	1	-1	0	0	0

Fructose-6-Phosphate	0	1	-1	0	0
Fructose-1,6-BisPhosphate	0	0	1	-1	0
Glycerone-3-Phosphate	0	0	0	1	-1
Glucose	-1	0	0	0	0

And set $\mathbf{b_{sqr}}$ = { 0, 0, 0, 0, -2 } to see what is produced/consumed when 2 molecules of glucose is consumed by the system. The 0's in $\mathbf{b_{sqr}}$ represent no net production of phosphate bearing intermediates, and the -2 represents the two molecules of glucose consumed.

Now:

$$R_{g5sqr} \cdot \mathbf{v} = \mathbf{b_{sqr}} = \{ 0, 0, 0, 0, -2 \}$$

LU decomposition of R_{g5sqr} subsequently allows for the solution:

$$\mathbf{v} = \{ v_1, v_2, v_3, v_4, v_5 \} = \{ 2, 2, 2, 2, 2 \}$$

Back substituting **v** into

$$R_{g5rmt} \cdot \mathbf{v} = \mathbf{b_{rmt}}$$

Where the rectangular "remnant" matrix R_{g5rmt} is:

R_{g5Rmt}	**r1**	**r2**	**r3**	**r4**	**r5**
H+	1	0	1	0	0
ATP	-1	0	-1	0	0
ADP	1	0	1	0	0
Glyceraldehyde-3-Phosphate	0	0	0	1	1

Gives (by simple matrix multiplication): $\mathbf{b_{rmt}}$ = { 4, -4, 4, 4 } This means 4 ATP are consumed, and 4 H^+, 4 ADP, and 4 molecules of Glyceraldehyde-3-Phosphate are produced.

In this example, merely knowing which molecules are neither produced nor consumed at steady state and specifying how much glucose to consume gives all non-trivial reaction stoichiometries as the solution of the system of reactions represented by the square matrix R_{g5sqr}. Molecular stoichiometries can then be derived by left-multiplying **v** by the rectangular

matrix remnant R_{g5rmt}. This operation shows how many Glyceraldehyde-3-Phosphate molecules are produced, how many ATP it costs, and how much ADP and H^+ byproduct are expected. This is a very simple example, and it should be noted that the true power of this method emerges with large systems (e.g. the complete oxidative metabolism of glucose).

Calculating Molecular Stoichiometries of Oxidative Metabolism

In order to calculate P/O ratios for oxidative metabolism, glycolysis, pyruvate dehydration, the citric acid cycle, the ETC, ATPase, several other reactions, and all necessary transport processes must be encoded as described above. Solution of the system is achieved with the aid of an interactive computer program developed by the author. The core linear algebraic manipulations utilize an LU decomposition algorithm freely available from the National Institute of Standards and Technology (TNT: linear algebra module, NIST). The complete commented source code is included in Appendix I1.

The systems studied were designed around several variables: substrate oxidized, electron shuttle utilized, and the H/ATP ratio. Substrates analyzed include glucose, beta-hydroxybutyrate, malate, pyruvate, and succinate. The electron shuttles considered are the glycerophosphate shuttle, the malate-aspartate shuttle, and the NADH-DH complex. The value of the H^+/ATP ratio as a potentially unknown variable is also a consideration. Calculations were executed using both H^+/ATP ratios of 4 (a convincing measured result), and 13/3 (the suggested subunit ratio of 10/3 plus one H^+ for transport). See the Introduction for the discussion and justification of these values.

The formulation of the mathematical structures representing the systems examined is as follows. First, the system is modeled as a fraction of suspended, cell-free mitochondria. This design is implemented to provide better comparison to values reported in the literature (since measurements are usually done on cell-free systems). If instead the systems were designed as whole cells respirating at steady state, another set of channels and carriers allowing for transport between the cytosol and extracellular space would have to be taken into account. This would be another degree of freedom affecting the P/O ratio (as most transporters require energy) that is not represented in the experimentally obtained P/O measurements reviewed in the Introduction. In addition, all individual reaction stoichiometries and atomic compositions of all molecules must be explicitly known. For well-established reactions (e.g. the steps of glycolysis), two textbooks (15, 52) and two websites (www.biocyc.com, www.reactome.com) were checked for consensus to confirm coefficients of reactions. For more controversial reactions (e.g. those occurring at the ETC complexes), primary literature was extensively reviewed to summarize reaction mechanisms. Even after encoding reactions according to published values of coefficients, every one was checked for conservation of atomic species. This was achieved by a matrix multiplication

$A \cdot R = N$ where:

A is an m X n atomic matrix (rows = atomic species, columns = molecules)
R is an m X n reaction matrix (rows = molecules, columns = reactions)
N is an m X n net production matrix (rows = atomic species, columns = reactions)

Assuming all reactions are balanced, the matrix N should be comprised entirely of zeros. Otherwise implies that one of the encoded reactions is creating or destroying atoms (or charge). In the event of a non-zero value, the source of the error must be traced back to either

an incorrect atomic encoding of a molecule, or an unbalanced reaction equation. See Appendix I for a complete list of reactions and/or reaction summaries used in this analysis.

In solving a system of equations, the square matrix fed to the LU decomposition contains only rows corresponding to molecules whose relative rates of production and consumption can be unambiguously specified. Specifically, this is most often the rate at which the substrate is consumed, which is set to a value of -1, and the rates at which intermediates that are neither produced nor consumed in a steady state system (e.g. 1,3-bisphosphoglycerate, cis-aconitate, or cytochromes that are a fixed part of the ETC), which are set to a value of 0. In the event that a square matrix is still singular, other rows with specified net molecular production/consumption may be swapped into the square matrix (e.g. CO_2 production in a complete oxidation reaction).

The output of the program is the vector **v** containing the relative rates of reaction within the system. From this vector the number of moles of ATP produced in the complete oxidation of 1 mol of substrate, and the moles of O_2 consumed is readily attained. The P/O ratio is equivalent to the reaction rate of hydrolysis of cytosolic ATP (representing the total usable energy produced) divided by twice the rate of the O_2 consuming reaction (since the P/O ratio is traditionally *molecules* of ATP produced to *atoms* of Oxygen consumed) at Complex IV of the ETC.

Results

Solutions for the complete oxidative metabolism of five substrates, utilizing three different electron shuttles (or exclusive mitochondrial metabolism in the cases of pyruvate and succinate which have no cytosolic component) and two different values of the H^+/ATP (4 and 13/3) were obtained. A summary of the P/O ratios are listed in Table 3, and depicted graphically (assuming an H^+/ATP ratio of 13/3) in Figure 4.

The generic mechanisms for the catabolism of all of the examined substrates are shown in Figures 5a-e. These pathways are derived from the reaction stoichiometries calculated by solving individual systems of equations for each substrate. There are compartmental constraints that must be placed on each step in the overall oxidation pathway. Glucose, for example, always has a cytosolic component of oxidation (see Figure 5a) because glycolytic machinery does not exist in the matrix. Thus, every time glucose is the substrate being oxidized, the P/O ratio will be partially dependent on the mechanism of electron shuttling, and it is impossible to have a value reflecting purely mitochondrial metabolism (note there is no value under "no shuttle:" for glucose in Table 3). Conversely, some substrates have no cytosolic component to their metabolism, making their P/O ratios independent of any electron shuttling mechanism. For example, the pyruvate dehydration complex (the first step in pyruvate oxidation) only exists in the matrix (see Figure 5b). As a result, pyruvate may only be metabolized within the mitochondrial matrix. Also, the machinery responsible for the citric acid cycle only exists inside the mitochondria, forcing succinate to be exclusively oxidized within the matrix as well (see Figure 5c). Notice that only one P/O ratio can be calculated (at a given H^+/ATP ratio) for pyruvate and succinate, so there are no shuttle-dependent values listed in Table 3. Finally, some substrates have early steps in their

metabolism that can occur either in the cytosol or in the mitochondrion. Both β-OHbutyrate and Malate have NADH-producing reactions (catalyzed by β-OHbutyrate dehydrogenase, and malic enzyme, respectively) that may occur in either compartment. Thus, these two substrates can give rise to either shuttle-dependent or shuttle–independent values as listed in Table 3.

Calculated P/O values range from 2.711 to 3.183 (for an H^+/ATP ratio of 13/3), or 3.000 to 3.500 (for an H^+/ATP ratio of 4/1) depending on substrate oxidized and shuttle utilized. Thus, the P/O ratios calculated for the specific combinations of substrate and electron shuttle shown in Tables 3a & 3b can vary by more than 17%. These estimates are fully contained within the range of all possible measured values cited in the introduction (1.07 - 3.73) but are higher than the commonly accepted theoretical values of ~2.5 and ~1.5 for NADH and $FADH_2$ linked substrates, respectively.

When separated by substrate, calculated P/O ratios are in the range of, but on average higher than, measured P/O ratios in the literature with one exception. The mid-range of measured values of malate was higher than the mid-range for calculated outputs presented here (but only for an H^+/ATP of 13/3, not when calculated with an H^+/ATP of 4/1; see Figure 6). The measured P/O ratios for malate generally exceeded the calculated values here because of additions to the experimental media. In these experiments, malate is often added with glutamate, a tradition started by Cross et. al. in 1949 (8) that is often continued today (16, 20, 53). Glutamate, though not studied in this analysis (and never metabolized by any of the systems analyzed), tends to inflate measured P/O ratios as a result of being simultaneously

consumed with a component of substrate level phosphorylation (13). This is similar to the way that glucose might inflate the P/O ratio of pyruvate if the two were mixed in a media, compared with pyruvate alone.

Also, the calculated P/O ratios for oxidation of succinate were much higher than literature values. The reason for this is the addition of inhibitors to the experimental media in succinate preparations. As briefly mentioned in the introduction, it was established early on that measuring the P/O ratio of succinate oxidation "required" the use of a site I inhibitor to prevent getting values that were "too high" (54). Modern preparations always include equimolar concentrations of rotenone (a complete Complex I inhibitor) in succinate-rich media (12, 16, 17, 18, 55).

When different H^+/ATP ratios are used to calculate the P/O ratios of a single substrate across all electron shuttles, or the P/O ratios of a given electron shuttle across all substrates studied, the shift is linear. [R^2=1 for 9 analyses, including lactate and acetoacetate as substrates, though they have been excluded from this work as they are not used in experimental P/O measurement designs in the literature (analyses not shown)] This result is expected (even trivial) as the underlying mathematical structure is a linear system of equations, and only changing one variable should yield a linear shift in the P/O ratio. This has an important implication, however. Relative variations of P/O ratios dependent upon which substrate is being oxidized, or which electron shuttle is being utilized, is not dependent on the value of the H^+/ATP ratio (for which a definite value is not available in the literature). That is, even if the absolute values attained with the calculations made here are incorrect

because the H^+/ATP ratio is not correct, the conclusion that the P/O ratio is variable by as much as 17% given the conditions analyzed still stands.

There is a clear trend dependent upon the electron shuttle implemented. In the complete oxidative metabolism of any of the substrates examined, P/O values are increasingly higher when the glycerophosphate, the malate-aspartate, and then the NADH-DH shuttles are used, respectively. The glycerophosphate shuttle is the least efficient because of its functional conversion of a cytosolic NADH to a matrix $FADH_2$. Although the other two shuttles convert cytosolic NADH to matrix NADH, there are efficiency differences because of differences in transport requirements (2 protons for MAS and 0 for the NADH-DH).

There is also a clear trend dependent upon the substrate metabolized. P/O values are increasingly higher when beta-hydroxybutyrate, succinate, pyruvate, malate, or glucose is metabolized, respectively within each shuttle. In general, the substrate-dependent differences in P/O ratios result from producing and consuming more electron equivalents (i.e. $FADH_2$ and NADH) as intermediates per oxygen atom consumed by the system. In the case of glucose, however, the P/O ratio is also bolstered by substrate-level phosphorylation.

Discussion

Over time, numerous studies in the literature have moved toward embracing two currently accepted theoretical P/O ratios: 2.5 for NADH-linked substrates and 1.5 for $FADH_2$-linked substrates (12). This assumes an H^+/ATP ratio of 4:1. However, if the H^+/ATP ratio is 13/3, the mechanistic P/O ratios should be 2.3 and 1.4 (13). Calculating a theoretical P/O ratio serves as a target or check to verify which of the disparate measured P/O values (often attributed to measurement error) should be recognized as correct. However, it seems no one has critically examined whether or not the theoretical calculations themselves are appropriate and correct. Besides, even though it may be possible to calculate a P/O ratio for a molecule of NADH or a molecule of $FADH_2$ as a substrate, the exact bearing of these numbers on P/O ratios of oxidizing organic substrates (e.g. succinate as an $FADH_2$–linked substrate) in a real system is unclear. Nor is it clear whether or not there is one, two, or any finite number of theoretical P/O values that empirical measurements should reflect. In fact, Kingsley-Hickman et. al. have suggested that experimentally measured P/O ratios and theoretically calculated P/O ratios should be considered entirely separately (56).

The Relationship Between Measured and Theoretical P/O Values

First, it is important to recognize that real measurements of P/O values may never match theoretical values. Aside from human errors in measurement, several phenomena may prevent respiratory machinery from functioning at theoretic values in vitro/vivo. Anything that effectively sinks the electrochemical gradient across the inner mitochondrial membrane

(thus undermining the ability of the ATPase to generate high-energy phosphate bonds) will reduce the P/O ratio. This process, generally referred to as slip (failure of an ETC complex to translocate protons despite successfully transferring a pair of electrons through the chain) or leak (proton movement with the gradient, not associated with an ETC complex) is reviewed in (57) and more recently in (58). Also, any process unrelated to energy metabolism occurring simultaneously at the time of measurement may alter the P/O ratio. If such a reaction requires ATP, NADH, $FADH_2$, or the proton gradient (as many processes required to maintain steady state and execute cellular functions do) in order to proceed, the observed P/O ratio will be decreased. If a net amount of protons are either consumed in the cytosol or produced in the matrix from ongoing reactions (whether related to metabolism or not), a measured P/O ratio of this system will again be less than the calculated theoretical maximum. Contrary to these conditions, if a real system is utilizing any anaerobic means of energy metabolism, the measured P/O ratio will be *higher* than the theoretical value.

Is the Current Calculation Correct?

In the well-oxygenated systems studied for P/O measurements, extensive anaerobic energy production is unlikely. However, the potential inefficiencies mentioned above are likely to be present. Thus, it follows that calculated P/O ratios should be greater than or equal to (accurately) measured values. Instead, the opposite is reported in the literature. Modern measurements of P/O ratios are often higher than the accepted theoretical values of 2.5 (or 2.3) and 1.5 (or 1.4) such as 2.9 and 1.8 (17), 3.1-3.7 (NADH-linked only: 16), 2.7-2.9 and 1.6-1.8 (22). Though the possibility certainly remains that some of these may be over-measurements (in fact, the range of the Toth study extends past the calculated values of this analysis), to the author's knowledge, no modern measurements are significantly lower than

2.5 and 1.5 (and thus 2.3 and 1.4). This direct contradiction calls current methods of calculation into question, while the method outlined herein is reinforced by producing P/O ratios that are almost uniformly greater than or equal to currently measured values.

In fact, on close examination the currently accepted theoretical P/O ratios are not computed correctly. The existing method of calculation (detailed in the Introduction) is oversimplified. It ignores transport phenomena, other contributors/sinks to the proton gradient, and the fact that a naturally respiring system maintains an overall steady state.

Active transport processes between the cytosol and matrix are important energy consumers, requiring the use of the electrochemical gradient. This reduces the cell's potential to generate ATPs. Some examples include (but are not limited to) the transport of pyruvate into the matrix, the transport of P_i into the matrix for GTP formation during TCA, and glutamate/aspartate exchange as part of the malate-aspartate shuttle.

Furthermore, careful attention must be paid to protons that are consumed or produced by all reactions, and the compartment in which those reactions occur. If a proton is consumed in the cytosol, there is one less proton available to power the ATPase, sinking the gradient. Similarly, if a proton is produced in the matrix, it will also lessen the driving force of the gradient. For example, when two acetyl-CoA's complete the citric acid cycle (as would occur in the metabolism of a single molecule of glucose), 4 protons are produced as a byproduct in the matrix. The net effect would be to nullify the driving force of up to 4 protons otherwise translocated into the cytosol by the ETC.

Finally, determining how many ATPs accumulate when NADH and $FADH_2$ are totally consumed by the ETC (as is done in the accepted calculation) does not reflect a steady state system. Ideally, a cell maintains constant levels of ADP, ATP, NADH, $FADH_2$, H^+, etc. and only consumes or produces substrate, water, oxygen, and carbon dioxide (fuels and waste products that are easily, and often freely, transported throughout the system). This means that energy equivalents are being consumed as they are produced and vice versa in a continuous cycle. While examining the steady state case adds the complication of necessarily considering the entire system simultaneously, it also lends itself to the mathematical tools used in this analysis. The benefit is instead of viewing energy metabolism as a cumbersome causal chain of events that lead to a massive accumulation of ATPs, this method facilitates a cyclic explanation. The cycle is such that every product of every reaction is simultaneously consumed as a reagent in the succeeding reaction. The result is a clean, circumscribed biochemical system that more closely approximates reality.

The hydrolysis of ATP (standard reaction linked to energy consuming processes) is represented by the following equation:

(cytosol) $H_2O + ATP^{4-} \rightarrow ADP^{3-} + P_i^{2-} + H^+$

From this, if ATP is being used at the same rate it is being produced, it is seen that there is an extra proton appearing in the cytosol every time an ATP is consumed/produced. This same proton disappeared from the matrix in the reverse reaction (ADP→ATP phosphorylation) that created the energy molecule at the inner membrane associated ATPase. Not considered

in contemporary literature calculations, this de facto translocated proton can potentially drive the production of more ATP. In fact, one way to envision ATP production and consumption is as a partially self-sustaining cycle (see Figure 7).

It is apparent from Figure 7 that at steady state, the overall H^+/ATP ratio can be seen as simply the F0/F1 ratio, and does not need to include a proton for transport. Instead, the proton consumed when ATP is produced in the matrix that is subsequently produced in the cytosol with ATP hydrolysis cancels the Pi transport proton (see Figure 7). Thus, the traditional notion of the H^+/ATP ratio outlined in the introduction should be reconsidered. In fact, this misconception may have been a source of confusion in the attempts (also reviewed in the Introduction) to measure H^+/ATP ratios, leading to differences qualified as 'including' or 'not including transport.' Some experimental designs may create systems that are closer to a natural steady state than others, altering the extent to which the self-sustaining portion of the cycle occurs. For example, many different buffering solutions have been used in experimental media, including varying concentrations of (to name just a few) Mg^{2+}, EDTA, and hexokinase (for "ATP trapping"). Even if ATP production were perfectly isolated, altered turnover and stability of produced ATP could cause H^+/ATP (and thus P/O) measures to be variable. Furthermore, if the H^+/ATP ratio does not need to include transport, this reinforces the implication that currently accepted theoretical P/O ratios are likely to be underestimates. Again, this is supported by the fact that modern measurements tend to be higher than currently accepted calculated P/O ratios. It is important to recognize that "including transport" and "not including transport" is an arbitrary naming convention. That is, the H^+/ATP ratio can still be reported as 13/3 including transport, but as long as the

method outlined herein is being utilized, the “transport proton” is handled separately and is guaranteed to always be accounted for and appropriately balanced. Problems arise when carelessness in accounting leads to the loss or gain of a proton with each ATP production/consumption cycle.

Perfect examples of non-steady state systems are exactly the calculations touted in the literature: mechanistic P/O ratios for NADH and $FADH_2$ of 2.3 and 1.4 or 2.5 and 1.5 depending on whether the H^+/ATP ratio is 13/3 or 4/1, respectively. These systems can be represented by the following equations:

$$x\ NADH + y\ ADP + z/2\ O_2 \rightarrow x\ NAD^+ + y\ ATP + z\ H_2O \quad (P/O = 2.5 \text{ or } 2.3)$$
$$x\ FADH_2 + y\ ADP + z/2\ O_2 \rightarrow x\ FAD + y\ ATP + z\ H_2O \quad (P/O = 1.5 \text{ or } 1.4)$$

Although there may be an academic reason to describe what might be called mechanistic P/O ratios for NADH and $FADH_2$, there are two reasons why these equations do not reflect real physiological situations. First, it behaves as if the bulky nucleotide electron carriers NADH and $FADH_2$ are being transported around a cell, tissue, organ system, or entire body as a primary energy-supplying medium. The overall reactions imply that NADH and $FADH_2$ are delivered to the respiratory apparatus and metabolized to produce NAD^+ or FAD, respectively. Such a view would require elaborate transporters and molecular systems to handle these large molecules. This is clearly not the case. Instead, physiologic systems use organic substrates like those examined here (glucose, etc.) which are much easier to transport and have far cleaner, freely diffusible breakdown products (CO_2 and H_2O). If oxidation of an organic substrate is occurring in parallel (which it always is), all of the reactions associated with it must be considered in the P/O calculation (as outlined above), and by linking the two

oxidative processes, the NAD^+ and FAD byproducts become necessary inputs to feed the production of NADH and $FADH_2$. This eliminates the need to worry about systems that handle and transport the bulky redox pairs, as they may remain in the same compartment, constantly cycling back and forth.

Another non-steady assumption is that it is acceptable to accumulate ATPs while massive amounts of ADP vanish. Using the methods described herein, the ATP steady state problem can easily be handled, allowing the ATPs to be consumed as they are produced. Thus, it is possible to calculate pure mechanistic values corresponding to the complete oxidative metabolism of NADH and $FADH_2$ using this method also. It is noted that this reaction is not physiologically meaningful, and does not represent a steady-state reaction. However these values (for NADH and $FADH_2$, respectively) should be either 3.3 and 1.8 or 3.67 and 2 depending on whether the H^+/ATP ratio is 13/3 or 4/1, respectively. The balanced, many-reaction mechanistic systems reduce to the following overall equations:

1.0 NADH + 1.0 cH + 0.5 O2 --> 1.0 NAD^+ + 1.0 H2O (P/O = 3.67 or 3.3)
1.0 FADH2 + 0.5 O2 --> 1.0 FAD^+ + 1.0 H_2O (P/O = 2.00 or 1.8)

Again, the calculations made here for mechanistic NADH and $FADH_2$ P/O ratios contribute to the consistent demonstration that currently accepted theoretical values are underestimates.

Flawed Calculations Lead to Failed Predictions

Even if mechanistic P/O ratios for NADH and $FADH_2$ (by themselves) exist and can be determined, it is not immediately obvious how these values justify the terms NADH-linked and $FADH_2$-linked substrate. Clearly, in a real system executing the complete oxidative

metabolism of an organic substrate (such as glucose), both NADH and $FADH_2$ will be produced and consumed as intermediates. Thus, the assumption that a P/O ratio associated with glucose will have one of the two accepted values instead of some intermediate value should be incorrect. That is, even disregarding every subtle complexity of calculating a P/O ratio cited above (e.g. substrate-level phosphorylation, side reactions, proton slip, etc.), the P/O ratio of a substrate should still be in some range between the two currently accepted values, and not simply one or the other.

Succinate is a perfect example. Because of the widely accepted dual P/O ratio system, the oxidation of succinate is expected to proceed with a P/O ratio of 1.5. With near uniformity, measured values reported in the literature significantly exceed this value. In addition to the fact that 1.5 has been shown to most likely be an underestimate, literature values are also low because ETC inhibitors are often added with succinate (12, 16, 17, 18, 55). Moreover, the system is sometimes controlled such that it only proceeds one step to fumarate (e.g. 59, 11). This might be acceptable as a strategy for determination of the P/O of $FADH_2$, however it is blatantly unrelated to a natural system that is freely metabolizing an excess of succinate as a fuel source. Despite all attempts to force control over this system, the reported values are often still high. This is because either the value of P/O for $FADH_2$ is higher than believed (as above), there is further downstream metabolism of fumarate and/or other unaccounted sources of phosphorylation, or both.

Putting the above aside, assuming P/O ratios for NADH and $FADH_2$ exist and are known, it is still not clear how these should be used to inform the calculation of a P/O ratio

for whole substrate metabolism. Without a rigorous mathematical infrastructure to bolster confidence that consideration is given to every intricate detail, it seems making such a calculation would be difficult. For example, in a landmark review Hinkle (12) used P/O ratios of 2.5 and 1.5 for NADH and $FADH_2$, respectively, to calculate the theoretical ATP yield given by the complete oxidation of glucose. He concludes the value is either 29.5 ATP or 31 ATP (depending on whether the G3P or MAS is utilized), challenging and displacing the previously accepted measured value, 36. Indeed, the ~30 ATP result is the most commonly cited value in textbooks. (14, 15) Presented here is the extended reasoning and calculation (relying upon an H^+/ATP ratio of 4), unaltered, from Hinkle et al. (12):

> Finally, the traditional calculation of the number of moles of ATP synthesized during the oxidation of 1 mol of glucose should be reconsidered. The complete oxidation of 1 mol of glucose yields 8 mol of matrix NADH which on oxidation would yield 20 mol of ATP, 2 mol of succinate yielding 3 mol of ATP, 2 mol of cytoplasmic NADH yielding 3 mol of ATP via the glycerol phosphate shuttle, 2 mol of cytoplasmic ATP from substrate level phosphorylation, and 2 mol of matrix GTP from succinyl-CoA synthetase. The matrix GTP forms ATP by nucleoside diphosphokinase. However, since the ATP must still be transported to the cytoplasm transporting 1 proton for each ATP, the amount of ATP that can be synthesized by oxidative phosphorylation is decreased by 2 protons or 0.5 ATP. Thus, the overall yield of ATP from glucose oxidation is 29.5 ATP per glucose, rather than the traditional value of 36 ATP per glucose based on integer values of the P/O ratios. If cytoplasmic NADH is oxidized via the malate-aspartate shuttle, then 4.5 ATP would be synthesized during oxidation of the 2 mol of cytoplasmic NADH, because glutamate/aspartate exchange is coupled to the influx of 1 proton per glutamate (LaNoue & Schoolworth, 1979), and the overall yield would be 31 ATP per glucose.

While more attention to detail is granted than most would give (i.e. considering the electron shuttling mechanism, some transport phenomena, and substrate phosphorylation), it is still an oversimplification, relying on inspection. Once again the ATP yield of a single molecule of glucose should be called into question and its calculation challenged by a more mathematically rigorous technique.

The currently accepted algorithm for this calculation (heretofore referred to as Hinkle Inspection) is also carried out for P/O ratios of 2.3 and 1.4, corresponding to the mechanistic

NADH and $FADH_2$ values of a system with an H^+/ATP ratio of 13/3 (as suggested by Hinkle in (13)). These values are compared with the calculated ATP yields from this study, and the Hinkle Inspection algorithm is explicitly reported (see Table 4). The ATP yields calculated in this analysis are consistently ~30% greater than those derived by Hinkle Inspection. While there are no guarantees about maintaining a balanced steady state or total energy accounting with Hinkle Inspection, assurance of those conditions is automatic in using the linear algebraic method on a deterministically modeled system.

The methodology presented here can also be used to delineate biochemical mechanisms and reaction pathways and circuits (as shown briefly with the schematic metabolism summaries of substrates listed in Results). A diagram representing this circuit can easily be translated from the model's output. One such diagram of a single analysis (complete oxidative metabolism of glucose, exclusively utilizing the MAS, with an H^+/ATP ratio of 13/3) is included as a demonstration (see Figure 8). Though it is a very complicated diagram, close inspection will reveal that the only molecular species either consumed or produced reduce to the following equation:

$$1\ C_6H_{12}O_6 + 6\ O_2 \rightarrow 6\ CO_2 + 6\ H_2O + \Delta H$$

A single molecule of glucose consumes six molecules of oxygen to produce six molecules of both carbon dioxide and water as byproducts, and in the process, a certain quantity of energy is stored and released via the production and consumption of ATP (the ΔH term, note: no net ATP is produced or consumed).

Striving for THE P/O Ratio—a Vain Pursuit?

In the past, two P/O ratios have been argued as standards against which measurements should be compared. In addition to demonstrating that these currently accepted theoretic P/O values are most likely wrong (underestimates), this analysis also shows that regardless of the absolute quantitative accuracy of this method, variability of the P/O ratio is inevitable. In a controlled experiment, the measured result should depend upon several factors: substrate, electron shuttle utilization, proton slip and leak, and any side reactions that may be occurring in the system. This is not a surprising conclusion. In a quick thought experiment, one can imagine an in vivo system in an anaerobic state. If this system was producing ATP exclusively through glycolysis or lactic acid fermentation, for example, no oxygen would be consumed and an infinite P/O ratio would be expected. Conversely, if one were studying the energetic metabolism of brown fat, the excess of UDP-1 (an uncoupling protein causing dose-dependent slip of the proton gradient) should make the measured P/O ratio arbitrarily lower than the theoretical value. Thus, it seems the conceivable natural range of the P/O ratio is from zero to infinity. This is the reason multiple groups using different methodologies cannot agree on a single pair of P/O ratios for energy metabolism: because no single value can possibly describe the infinite permutations of varying conditions.

In this analysis, P/O ratios were demonstrated to differ by as much as 17% (regardless of the H^+/ATP ratio), simply by altering which of five substrates was consumed and which of four electron shuttles was utilized (including ‘no shuttle’ as an option). Table 5 is included as a summary of variable conditions that may alter a P/O ratio. There are 15 that have been explicitly stated in this text, however others certainly exist that have not been mentioned. Note that the calculations in this work are idealized (excluding all inefficiencies) within a

purely theoretical framework in which all but two conditions are identical. This would suggest that in an experimental setting, where inefficiency does exist and more than two of these conditions are varied by the sheer design of the system preparation, one should expect measured P/O ratios to vary much more widely than seen here. As such, it is no surprise that an overall range of 1.07 to 3.73 (from all studies reviewed) is observed.

Perhaps the notion of a variable P/O ratio should be embraced, and future experimentation focused on exploring the basis of this variability. Instead of striving to find a single number to describe oxidative cellular machinery, a variable P/O can be used (in a standardized experimental system) as an indicator of oxidative efficiency for a given set of variables and conditions. After all, a variable P/O ratio has useful physiological implications. Different tissues have different metabolic goals, and thus employ slight variations of energy production, giving rise to different P/O ratios (60). For example, enzymatic compartmentalization may be tissue specific out of mechanistic need. Very early on it was shown that the P/O ratios for rat heart muscle vs. liver differ in the face of otherwise identical experimental conditions and supporting media (54). The same study showed that guinea pig myocyte P/O ratios were significantly greater than those of rat myocytes, again in the setting of identical conditions. More recently it was shown that the heart may prefer ketones and fatty acids to glucose as its primary substrate (61).

This analysis demonstrates that different electron shuttles are unquestionably associated with different P/O ratios. This may be an important point of in vivo metabolic control. For example, Scholz et. al. demonstrated that thyroxin (T3) can alter the balance of G3P to MAS

utilization in cardiomyocyte and hepatocyte mitochondria in a tissue-specific fashion (62). This example of hormonal regulation of energetic efficiency operates by changing the P/O ratio via altering the extent and balance of shuttle utilization. Also, Cairns et. al. (60) have suggested, though a thermodynamic argument, that temporal efficiency may also be a factor. While the liver attempts to maximize chemical efficiency in terms of ATPs per Oxygen consumed, the brain may be trying to turnover ATP at a maximum efficiency per unit of time, and the heart is simply maximizing the number of ATPs it can produce. This suggests that measures of oxidative efficiency other than P/O ratio be considered and tested experimentally.

Summary of Discussion

In recent years, consensus in the literature surrounding the P/O ratio has moved toward two calculated theoretical values (2.5 for NADH-linked substrates, and 1.5 for $FADH_2$-linked substrates) in an attempt to discover which of many varied measurements are the closest approximations to true values. The analysis outlined here demonstrates the following:

1) Theoretical P/O ratios have been inappropriately calculated to date, and are likely underestimates. This is further demonstrated by failed predictions of ATP production.

2) Assuming the ATPase subunit ratio hypothesis of the H^+/ATP ratio is correct (and that ratio is 10:3) a steady state, respiring mitochondrion that is exclusively using oxidative metabolism will have a theoretical maximum efficiency (assuming no slip, leak, proton uncoupling, or side reactions), in terms of a P/O ratio, in the range of

2.767 – 3.238 for any combination of the five substrates and four electron shuttles studied.

3) Mechanistic P/O ratios are not very useful tools in approaching real physiological conditions. Despite this fact, the mechanistic P/O ratio for NADH oxidation should be either 3.3 or 3.67 instead of the currently accepted 2.3 or 2.5 (depending whether the H^+/ATP ratio is 13/3 or 4/1, respectively), and the mechanistic P/O ratio for $FADH_2$ should be 1.80 or 2.00 instead of the accepted values 1.4 and 1.5 (with similar H^+/ATP dependence).

4) The P/O ratio should be variable, its value based on many conditions.

References

1. Kalckar, H. 1937. Phosphorylation in kidney tissue. *Enzymologia* 2: 47-52.

2. Ochoa S. 1941. "Coupling" of phosphorylation with oxidation of pyruvic acid in brain. J. Biol. Chem. 138: 751-773.

3. Belitser VA., Tsibakowa ET. 1939. The mechanism of phosphorylation associated with respiration. *Biokhimiya* 4: 516-534.

4. Ochoa S. 1943. Efficiency of aerobic phosphorylation in cell-free heart extracts. *J. Biol. Chem.* 151: 493-505.

5. Devlin, Thomas M. (ed) 2001. Textbook of Biochemistry with Clinical Correlations (5th ed.). John Wiley + Sons, Inc.; Hoboken, NJ : 1248 pp.

6. Lehninger A., Smith S. 1949. Efficiency of oxidative phosphorylation coupled to electron transport between dihydrodiphosphopyridine nucleotide and oxygen. *J. Biol. Chem.* 181: 415-429.

7. Bartley W. 1953. Efficiency of oxidative phosphorylation during the oxidation of pyruvate. *Biochem. J.* 54(4): 677-682.

8. Cross R., Taggart J., Covo G., Green D. 1949. Studies on the cyclophorase system: VI. The Coupling of Oxidation and Phosphorylation. *J. Biol. Chem.* 177: 655-678.

9. Mitchell P. 1961. Coupling of phosphorylation to electron and hydrogen transfer by a chemi-osmotic type of mechanism. *Nature* 191: 144-148.

10. Mitchell P. 1970. Aspects of Chemiosmotic Hypothesis. *Biochem. J.* 116(4): 5P-6P.

11. Stoner C. 1987. Determination of the P/2e- Stoichiometries at the Individual Coupling Sites in Mitochondrial Oxidative Phosphorylation. *J. Biol. Chem.* 262: 10445-10453.

12. Hinkle P., Kumar M., Resetar A., Harris D. 1991. Mechanistic Stoichiometry of Mitochondrial Oxidative Phosphorylation. *Biochemistry* 30: 3576-3582.

13. Hinkle P. 2005. P/O ratios of mitochondrial oxidative phosphorylation. *Biochim. Biophys. Acta* 1706: 1-11.

14. Nelson, David L., Cox, Michael M. 2004. Lehninger Principles of *Biochemistry.* W.H. Freeman; New York, NY : 1100 pp.

15. Stryer, Lubert 1995. Biochemistry (4th ed.). W.H. Freeman; New York, NY : 1064 pp.

16. Toth P., Sumerix K., Ferguson-Miller S., Suelter C. 1990. Respiratory Control and ADP:O Coupling Ratios of Isolated Chick Heart Mitochondria. *Arch. Biochem. Biophys.* 276(1): 199-211.

17. Lee C., Gu Q., Xiong Y., Mitchell R., Ernster L. 1996. P/O Ratios Reassessed: Mitochondrial P/O Ratios Consistently Exceed 1.5 with Succinate and 2.5 with NAD-linked Substrates. *FASEB J.* 10: 345-350.

18. Gnaiger E., Mendez G., Hand S. 2000. High Phosphorylation Efficiency and Depression of uncoupled respiration in mitochondria under Hypoxia. *Proc. Natl. Acad. Sci. U.S.A.* 97: 11080-11085.

19. Lemasters JJ., Grunwald R., Emaus RK. 1984. Thermodynamic Limits to the ATP/Site Stoichiometries of Oxidative Phosphorylation by Rat Liver Mitochondria. *J. Biol. Chem.* 259(5): 3058-3063.

20. Lemasters JJ. 1984. The ATP-to-oxygen stoichiometries of oxidative phosphorylation by rat liver mitochondria. An analysis of ADP-induced oxygen jumps by linear nonequilibrium thermodynamics. *J. Biol. Chem.* 259(21): 13123-13130.

21. Beavis A., Leninger A. 1986. The upper and lower limits of the mechanistic stoichiometry of mitochondrial oxidative phosphorylation. *Eur. J. Biochem.* 158: 315-322.

22. Beavis A., Leninger A. 1986. Determination of the upper and lower limits of the mechanistic stoichiometry of imcompletely coupled fluxes.. *Eur. J. Biochem.* 158: 307-314.

23. Villalobo A., Lehninger A. 1979. The proton stoichiometry of electron transport in Ehrlich ascites tumor mitochondria. *J Biological Chem* 254(11): 4352-4358.

24. Pozzan, T., Di Vigilio F., Bragadin, M., Miconi V., Azzone GF. 1979. H+/site, charge/site, and ATP/site ratios in mitochondrial electron transport. *Proc. Natl. Acad. Sci. U.S.A.* 76(5): 2123-2127.

25. Villalobo A., Lehninger A. 1980. Stoichiometry of H+ Ejection Coupled to Electron Transport through Site 2 in Ascites Tumor Mitochondria. *Arch. Biochem. Biophys.* 205(1): 210-216.

26. Villalobo A., Alexandre A., Lehninger A. 1984. H+ Stoichiometry of Sites 1 + 2 of the Respiratory Chain of Normal and Tumor Mitochondria. *Arch. Biochem. Biophys.* 233(2): 417-427.

27. Trumpower, B. 1990. The Protonmotive Q Cycle. *J. Biol. Chem.* 266(20): 11409-11412.

28. Antonini G., Maltasta F, Sarti P., Brunori M. 1993. Proton Pumping by cytochrome oxidase as studied by time-resolved stopped-flow spectrophotometry. *Proc. Natl. Acad. Sci. U.S.A.* 90: 5949-5953.

29. Thelen M., O'Shea P., Petrone G., Azzi A. 1985. Proton Translocation by a Native and Subunit III-depleted Cytochrome c Oxidase Reconstituted into Phospholipid Vesicles. *J Biological Chem* 260(6): 3626-3631.

30. Galkin, AS., Grivennikova, VG., Vinogradov, AD. 1999. .->H+/2e- stoichiometry in NADH-quinone reductase reactions catalyzed by bovine heart submitochondrial particles. *FEBS Lett.* 451: 167-161.

31. LaNoue, K., Schoolwerth, A. 1979. Metabolite transport in mitochondria. *Ann. Rev. Biochem.* 48: 871-922.

32. Klingenberg M., Rottenberg, H. 1977. Relation between the Gradient of the ATP/ADP Ratio and the Membrane Potential across the Mitochondrial Membrane. *Eur. J. Biochem.* 73: 125-130.

33. Mitchell P., Moyle J. 1968. Proton translocation coupled to ATP hydrolysis in rat liver mitochondria. *Eur. J. Biochem.* 4(4): 530-539.

34. Thayer WS., Hinkle PC. 1973. Stoichiometry of adenosine triphosphate-driven protontranslocation in bovine heart submitochondrial particles. *J. Biol. Chem.* 248(15): 5395-5402.

35. Brand MD., Lehninger AL. 1977. H+?ATP ratio during ATP hydrolysis by mitochondria: modification of the chemiosmotic theory. *Proc. Natl. Acad. Sci. U.S.A.* 74(5): 1955-1959.

36. Ferguson SJ., Sorgato MC. 1982. Proton Electrochemical Gradients and Energy-Transduction Processes. *Annual Reviews in Biochemistry* 51: 185-217.

37. Berry EA., Hinkle PC. 1983. Measurement of the Electrochemical Proton Gradient in Submitochondrial Particles. *J. Biol. Chem.* 258(3): 1474-1486.

38. Woelders H., Zande WJ., Colen AM., Wanders R., Dam K. 1985. The phosphate potential maintained by mitochondria in State 4 is proportional to the proton-motive force. *FEBS Lett.* 179(2): 278-282.

39. Vink R., Bendall M., Simpson S., Rogers P. 1984. Estimation of H+ to Adenosine 5'-Triphosphate Stoichiometry of Escherichia coli ATP Synthase Using 31P NMR. *Biochemistry* 23: 3667-3675.

40. Cross RL., Duncan TM. 1996. Subunit rotation in F0F1-ATP synthases as a means of coupling proton transport through F0 to the binding changes in F1. *J. Bioenerg. Biomembr.* 28(5): 403-408.

41. Junge W. 1999. ATP synthase and other motor proteins. *Proc. Natl. Acad. Sci. U.S.A.* 96: 4735-4737.

42. Klingenberg M. 1979. The Ferricyanide Method for Elucidating the Sidedness of Membrane-Bound Dehydrogenases. *Meth. Enzymol.* 56: 229-233.

43. Klingenberg M., Buchholz M. 1970. Localization of the Glycerol-Phosphate Dehydrogenase in t heOuter Phase of the Mitochondrial Inner Membrane. Eur. *J. Biochem.* 13: 247-252.

44. LaNoue K., Tischler M. 1974. Electrogenic Characteristics of the Mitochondrial Glutamate-Aspartate Antiporter. *J. Biol. Chem.* 249(23): 7522-7528.

45. Meijer AJ 2003. Amino acids as regulators and components of nonproteinogenic pathways. *J. Nutr*. 133(6Sup1): 2057S-2062S.

46. Palmieri, F. 2004. The mitochondrial transporter family (SLC25): physiological and pathological implications. *Eur. J. Phys.* 447: 689-709.

47. Nohl H. 1987. Demonstration of the existence of an organo-specific NADH dehydrogenase in heart mitochondria. *Eur. J. Biochem.* 169: 585-591.

48. Schonheit K., Nohl H. 1996. Oxidation of Cytosolic NADH via Complex I of Heart Mitochondria. *Arch. Biochem. Biophys.* 327(2): 319-323.

49. Bonarius H., Schmid G., Tramper J. 1997. Flux analysis of underdetermined metabolic networks: The quest for the missing constraints. *Trends in Biotechnology* 15(8): 308-314.

50. Ramakrishna R., Edwards J., McCulloch A., Palsson B. 2001. Flux-balance analysis of mitochondrial energy metabolism: consequences of systemic stoichiometric constraints. *Am. J. Phys. Reg.* 280: 695-704.

51. Lay, David 2003. Linear Algebra and its Applications (3rd ed.). Addison Wesley; Boston, MA : 492 pp.

52. Champe, P., Harvey, R., Ferrier 2004. Biochemistry (3rd ed.). Lippincott Wilkin & Williams; Philadelphia, PA : 534 pp.

53. Devin A., Buerin B., Rioulet M. 1997. Control of Oxidative Phosphorylation in rat Liver Mitochondria: Effect of Ionic Media. *Biochim. Biophys. Acta* 1319: 293-300.

54. Greengard P., Minnaert K., Slater E., Betel I. 1959. Yield of Oxidative Phosphorylation associated with the oxidation of Succinate to Fumarate. *Biochem. J.* 73: 637-646.

55. Hinkle PC., Yu ML. 1979. The phosphorus/Oxygen Ratio of Mitochondrial Oxidative Phosphorylation. *J. Biol. Chem.* 254(7): 2450-2456.

56. Kingsley-Hickman PB., Sako EY., Mohanakrishnan P., Robitaille PM., From AH., Foker JE., Ugurbil K. 1987. 31P NMR studies of ATP synthesis and hydrolysis kinetics in the intact myocardium. *Biochemistry* 26(23): 7501-7510.

57. Brand M., Chien L., Ainscow E., Rolfe D., Proter R. 1994. The causes and functions of mitochondrial proton leak. *Biochim. Biophys. Acta* 1187: 132-139.

58. Kadenbach, B. 2003. Intrinsic and Extrinsic uncoupling of oxidative phosphorylation. *Biochim. Biophys. Acta* 1604: 77-94.

59. Copenhaver J., Lardy J. 1952. Oxidative phosphorylations: pathways and yield in mitochondrial preparations. *J. Biol. Chem.* 195: 225-238.

60. Cairns C., Walther J., Harken A., Banerjee A. 1998. Mitochondrial Oxidative Phosphorylation thermodynamic efficiencies reflect physiological organ roles. *Am. J. Phys. Reg*. 274: 1376-1383.

61. Ziegler A., Zaugg C., Buser P., Seelig J., Kunnecke B. 2002. Non-invasive measurements of myocardial carbon metabolism using in vivo 13C NMR spectroscopy. *NMR Biomed* 15: 222-234.

62. Scholz T., TenEych C., Schutte B. 2000. Thyroid Hormone Regulation of the NADH Shuttles in Liver and Cardiac Mitochondria. *J. Mol. Cell. Cardiol.* 32: 1-10.

Tables

Subs	Author	Year	P/O	Error
Glucose	Kalckar	1937	2	NR
Pyruvate	Belitzer	1939	2	NR
Pyruvate	Ochoa	1943	3.2	0.4
B-OHButyrate	Lehninger	1949	2.02	0.26
Succinate	Cross	1949	1.25	0.179
Malate + Glu	Cross	1949	2.25	0.1
Pyruvate	Cross	1949	2.47	0.12
Pyruvate	Bartley	1953	2.35	0.25
B-OHButyrate	Chance	1955	2.6	NR
Succinate	Chance	1955	1.8	NR
Succinate	Greenard	1959	1.91	0.29
B-OHButyrate	Hinkle	1979	2.11	0.13
Succinate	Hinkle	1979	1.39	0.1
Succinate	Lemasters	1984	1.85	0.23
Malate	Lemasters	1984	2.89	0.31
B-OHButyrate	Lemasters	1984	2.93	0.42
Succinate	Beavis	1986	1.71	0.12
B-OHButyrate	Beavis	1986	2.78	0.12
Glucose	Stoner	1987	2.62	0.05

Glucose + Pyruvate	Kingsley-Hickman	1987	2.34	0.38
Pyruvate + Malate	Toth	1990	3.43	0.3
B-OHButyrate	Toth	1990	3.21	0.24
Succinate	Toth	1990	1.9	0.01
B-OHButyrate	Hinkle	1991	2.27	0.08
B-OHButyrate	Hinkle	1991	2.85	0.15
Pyruvate + Malate	Lee	1996	2.73	0.22
Succinate	Lee	1996	1.81	0.07
Malate + Glu	Devin	1997	2.49	0.22
Succinate	Gnaiger	2000	1.77	0.04

Table 1. 29 studies reporting P/O measurements between 1937 and 2000 are summarized as ranges reported within the text of the reference. NR signifies no range reported, i.e. the work reports a single value.

Organism	Subunit ratio	Source
Yeast Mitochondria	10:3	Stock, 1999
E. Coli	10:3	Jian, 2001
Leaf Chloroplast	12:3	Turina, 2003
Leaf Chloroplast	14:3	Seelert, 2000
Cyanobacteria	12:3	Junge, 1999
Ilyobacter	11:3	Stahlberg, 2001
Enterococcus	7:3	Murata, 2003
Archaea	{6,8,12,13}:3*	Muller, 2004

* Archaea is a class of ancient bacteria. Here Muller et al. studied species for A1/A0 subunit ratios which varied according to alternately spliced transcripts

Table 2. 8 studies reporting F_0/F_1 subunit ratios of the mitochondrial ATPase in various species.

3a.

		No Shuttle:	Shuttle:		
	H+/ATP = 4.33	**Matrix**	**G3P**	**MAS**	**NADH**
Substrate:	**Glucose**	*	2.933	3.083	3.183
	β-OH Butyrate	2.844	2.711	2.811	2.878
	Malate	3.017	2.767	2.917	3.017
	Pyruvate	2.960	*	*	*
	Succinate	2.843	*	*	*

3b.

		No Shuttle:	Shuttle:		
	H+/ATP = 4.00	**Matrix**	**G3P**	**MAS**	**NADH**
Substrate:	**Glucose**	*	3.222	3.389	3.500
	β-OH Butyrate	3.148	3.000	3.111	3.185
	Malate	3.333	3.056	3.222	3.333
	Pyruvate	3.267	*	*	*
	Succinate	3.143	*	*	*

Table 3. Calculated P/O ratios organized by substrate consumed and electron shuttle utilized. 3a reports calculated values in a system where the H^+/ATP ratio was 13/3, and 3b contains values from a system where the H^+/ATP ratio was 4.

		Hinkle Inspection*	Linear Algebraic
H/ATP =4.000	G3P	29.50	38.67
	MAS	31.00	40.67
H/ATP =4.333	G3P	27.54	35.20
	MAS	28.88	37.00

Table 4. Calculated theoretical ATP yield for glucose: Hinkle inspection vs. Linear Algebraic Method

***Hinkle Inspection = [#NADH * (P/O_{NADH})] + [#$FADH_2$ * (P/O_{FADH2})] + SL_{phos} - [#H^+_{trans}/($H\text{-}ATP_{ratio}$)]**

Where:

#NADH = number of NADH produced #$FADH_2$ = number of $FADH_2$ produced
P/O_{NADH} = P/O_{ratio} of NADH (given H/ATP) P/O_{FADH2} = P/O_{ratio} of $FADH_2$ (given H/ATP)
SL_{phos} = #substrate-level phosphorylations #H+trans = number of protons used for xport
$H\text{-}ATP_{ratio}$ = the H^+/ATP ratio of the system

E.G. for G3P shuttle and H^+/ATP = 4.0 from the verbal argument in the block quote:
ATP Yield = (8*2.5) + (4*1.5) + 4 – (2/4.0) = 20+6+4-0.5 = **29.50**

1	Substrate
2	Electron Shuttle
3	H+/ATP Ratio (F0/F1 Subunit variability)
4	Substrate-level phosphorylation
5	Proton consuming/producing reactions of energy metabolism
6	Slip and/or leak of proton gradient
7	Energy consuming (ATP or proton gradient) side reactions
8	Side Redox reactions
9	Cell type
10	Media additions (inhibitors, Mg2+, EDTA, etc.)
11	Measurement techniques
12	pH
13	Temperature
14	Hormonal control
15	Organism

Table 5. Short List of Degrees of Freedom in Measuring a P/O Ratio.

Figures

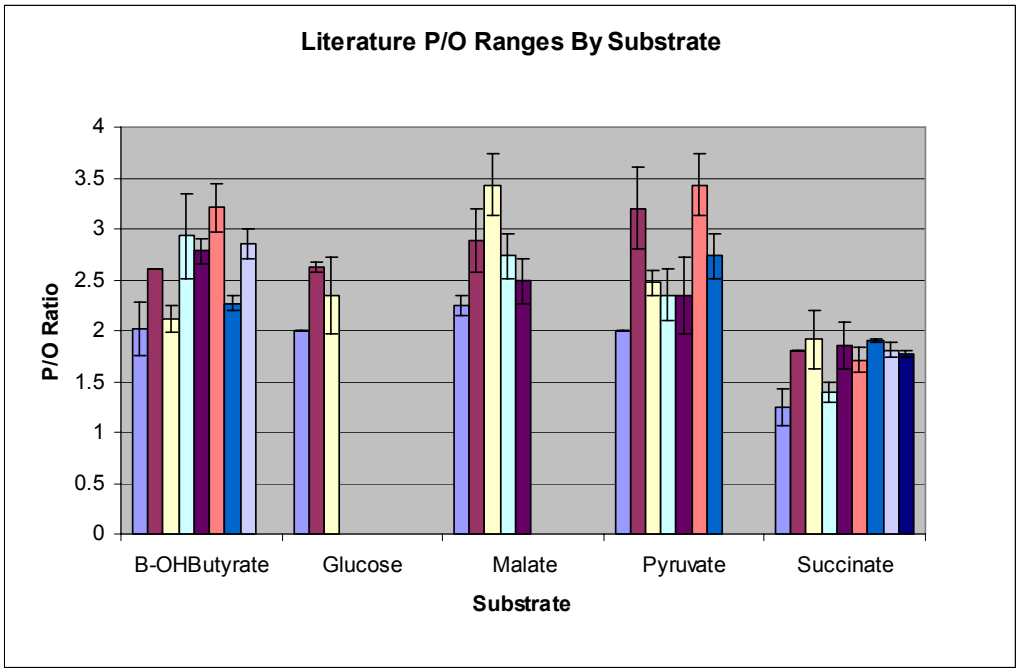
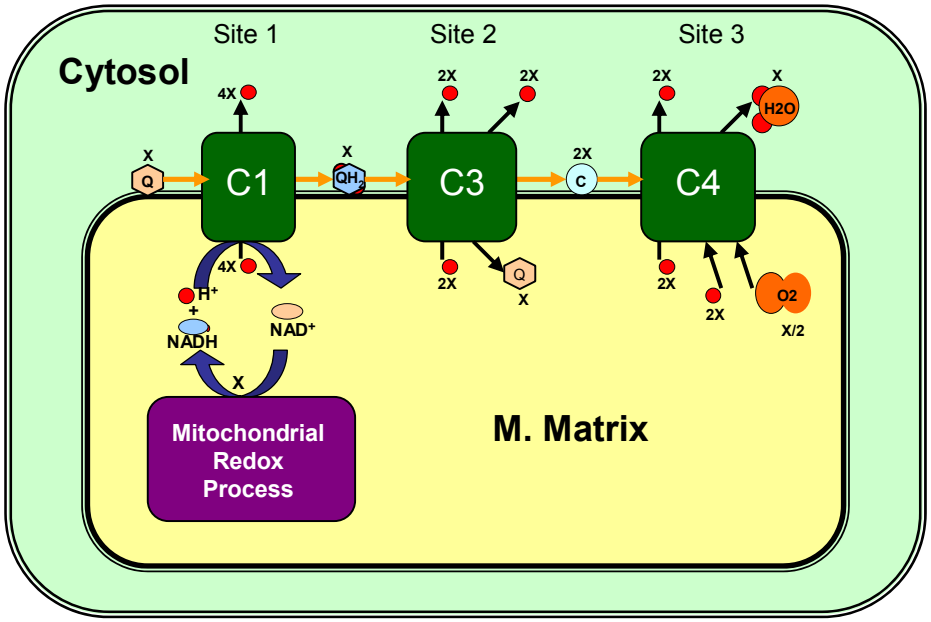


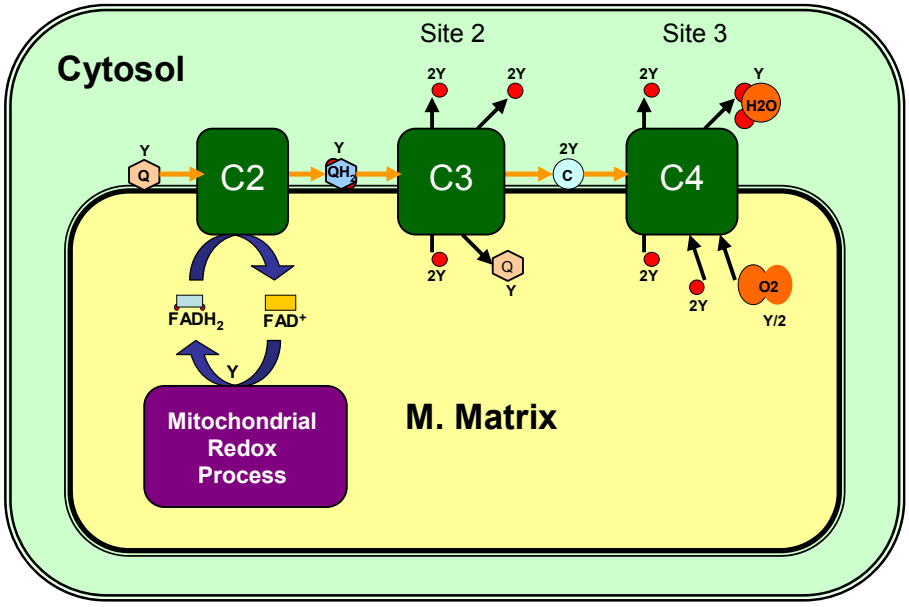
Figure 1. Summary of Measured P/O Values from Literature (29 sources). Bar height represents the middle of a reported range. Error bars represent the entire range reported regardless whether the range represented statistical error of one consensus value, several measurements, or otherwise (they do not reflect any statistical analysis).

2a.



$$11\ (m)\ H^+ + 1\ (m)\ NADH + 0.5\ (x)\ O_2 \rightarrow\ 10\ (c)\ H^+ + 1\ (m)\ NAD^+ + 1\ (x)H_2O$$

2b.



$$6\,(m)\,H^+ + 1\,(m)\,FADH_2 + 0.5\,(x)\,O_2 \rightarrow 6\,(c)\,H^+ + 1\,(m)\,FAD^+ + 1\,(x)H_2O$$

Figure 2. Summary of ETC mechanisms. 2a depicts NADH oxidation from Complex I to Complex IV. 2b depicts $FADH_2$ oxidation from Complex II to Complex IV. Summary reactions are provided.

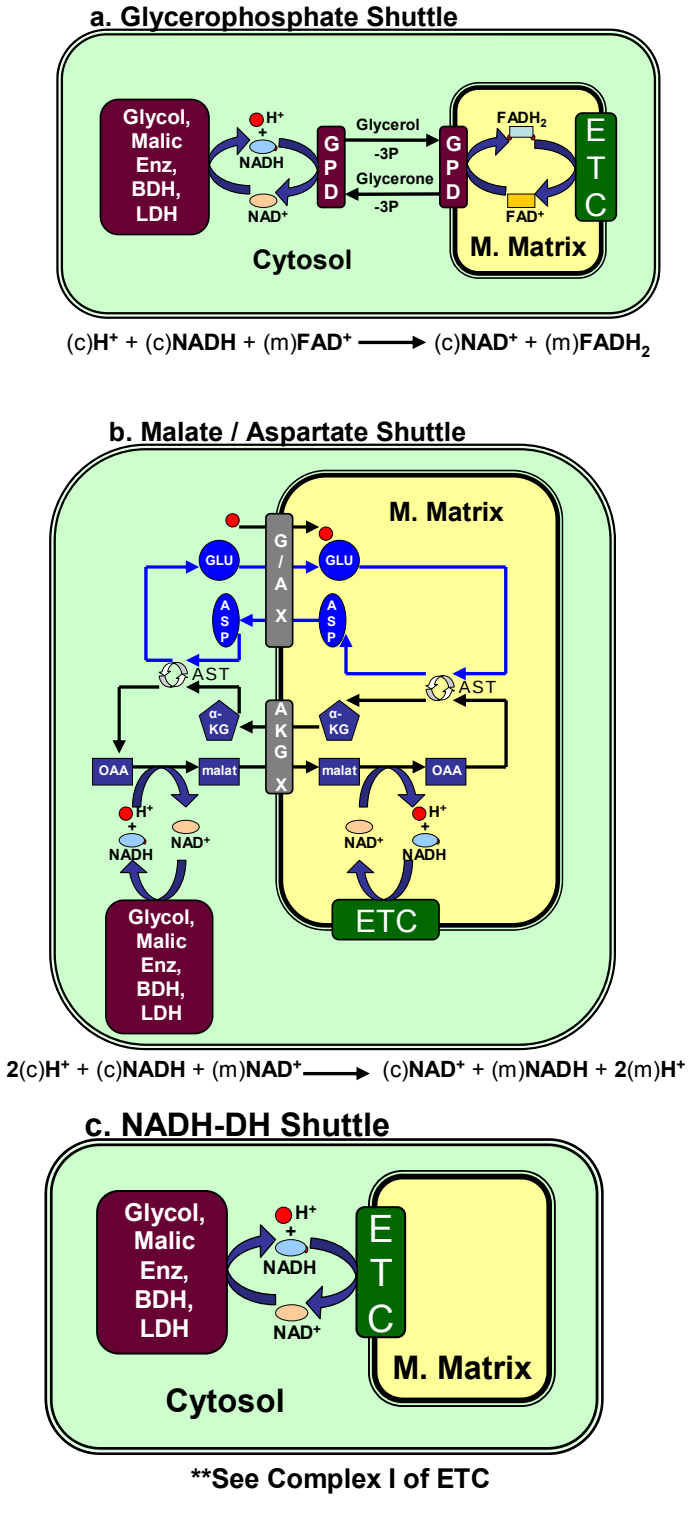


Figure 3. Summary of mechanisms for various electron shuttles. 3a is the glycerophosphate shuttle (G3P), 3b. is the malate/aspartate shuttle (MAS), and 3c is the NADH-DH complex.

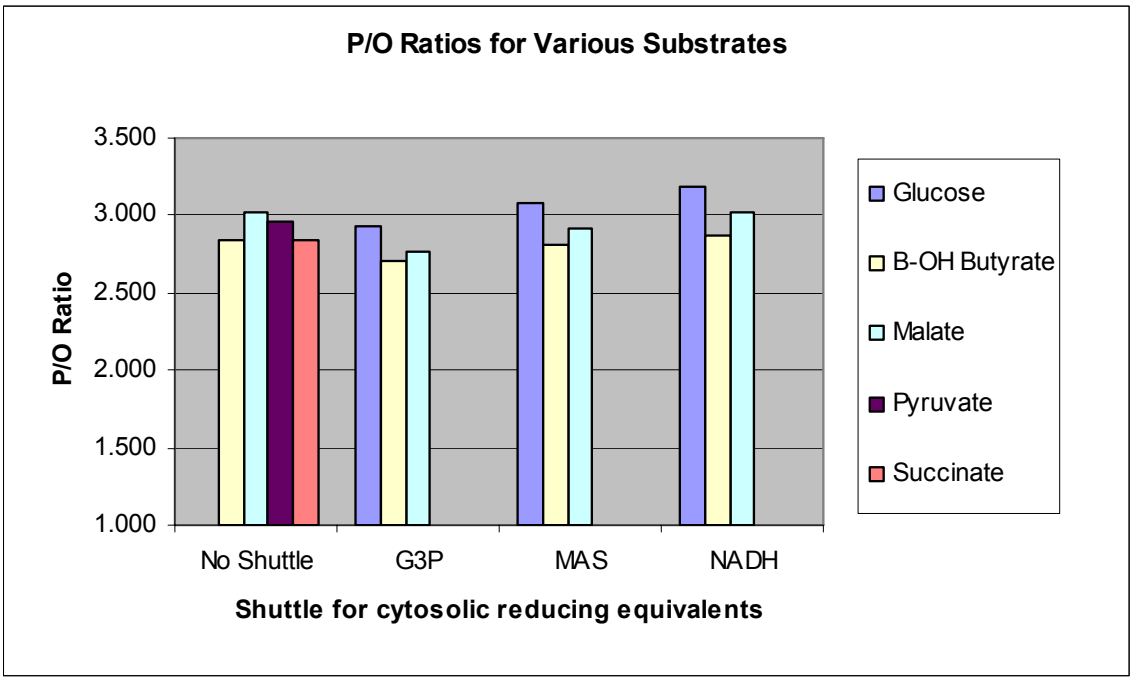
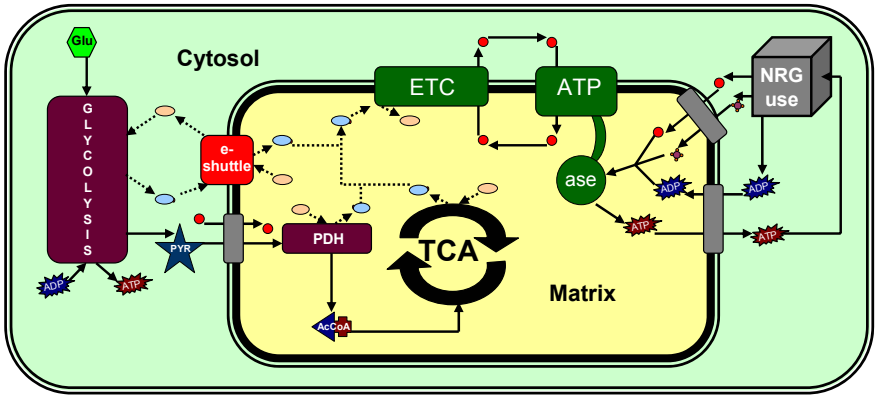


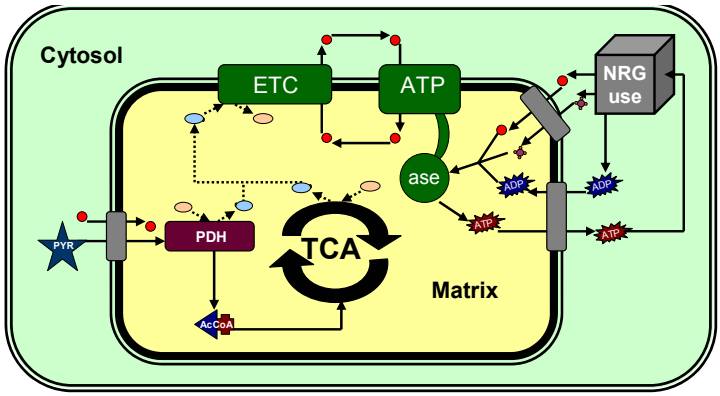
Figure 4. Summary of calculated P/O Outputs

5a. Glucose Metabolism



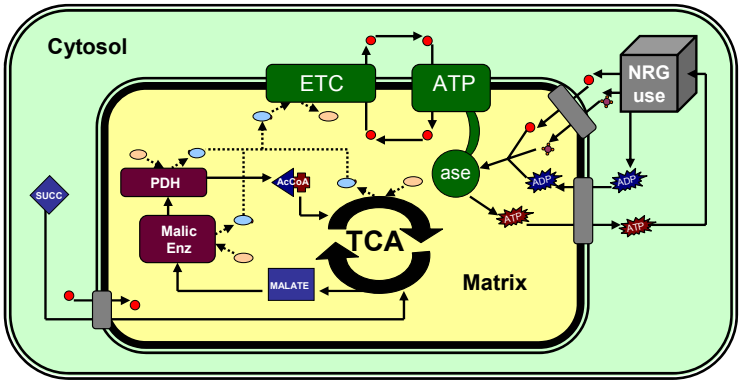
(c)Glucose + 6 O_2 --> 6 CO_2 + 6 H_2O

5b. Pyruvate Metabolism



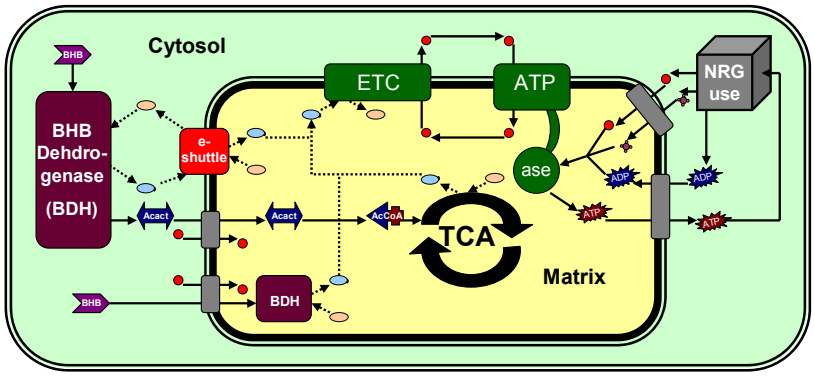
(c)Pyruvate + 2.5 O_2 + (c)H^+ --> 3 CO_2 + 2 H_2O

5c. Succinate Metabolism



(c)Succinate + 3.5 O_2 + 2 (c)H^+ --> 4 CO_2 + 3 H_2O

5d. β-OHButyrate Metabolism



(c)β-OHButyrate + 4 O_2 + 1 (c)H^+ --> 4 CO_2 + 3 H_2O

5e. Malate Metabolism

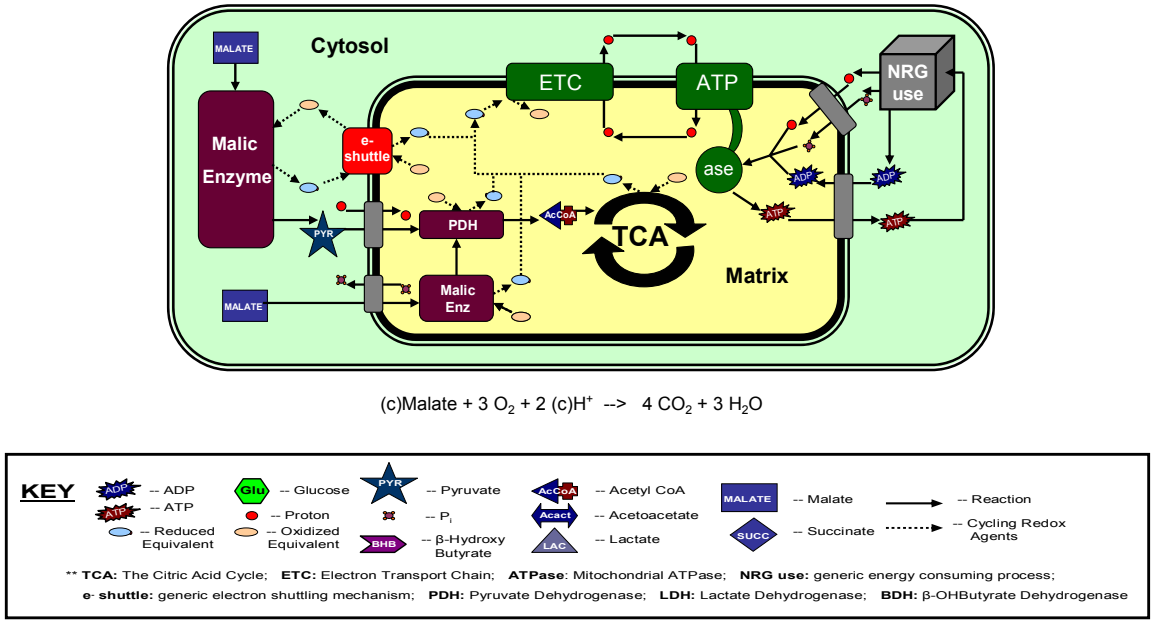


Figure 5. Mechanisms for complete oxidative metabolism of: 5a. glucose; 5b. pyruvate; 5c. succinate; 5d β-OHbutyrate; 5e malate.

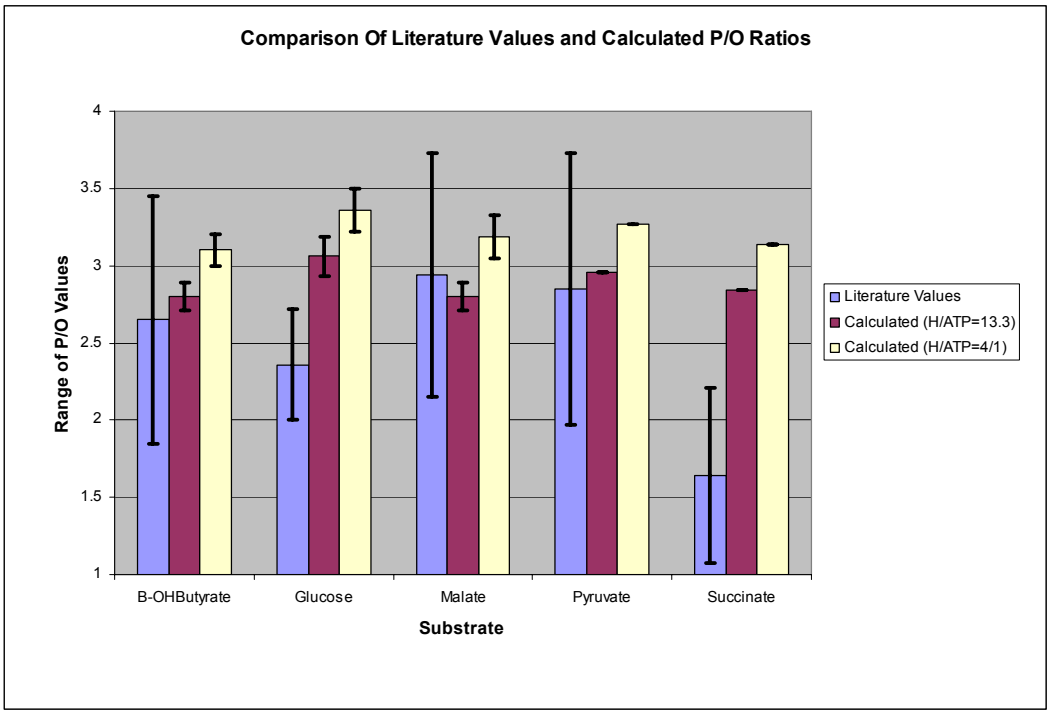
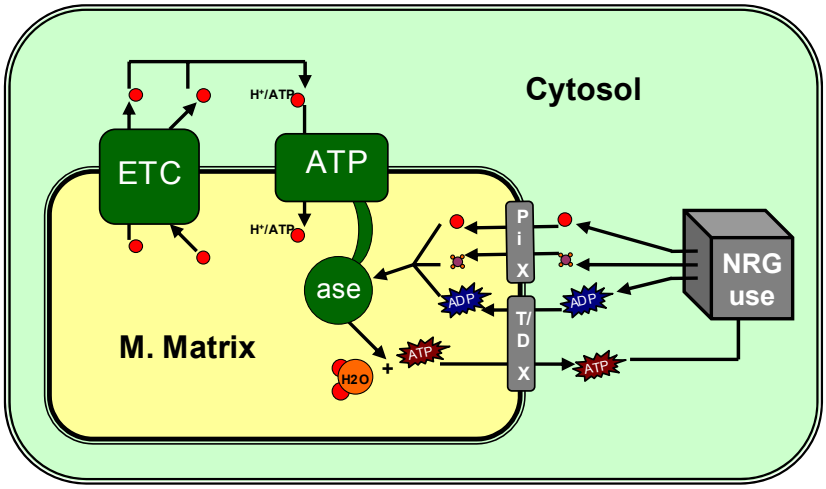
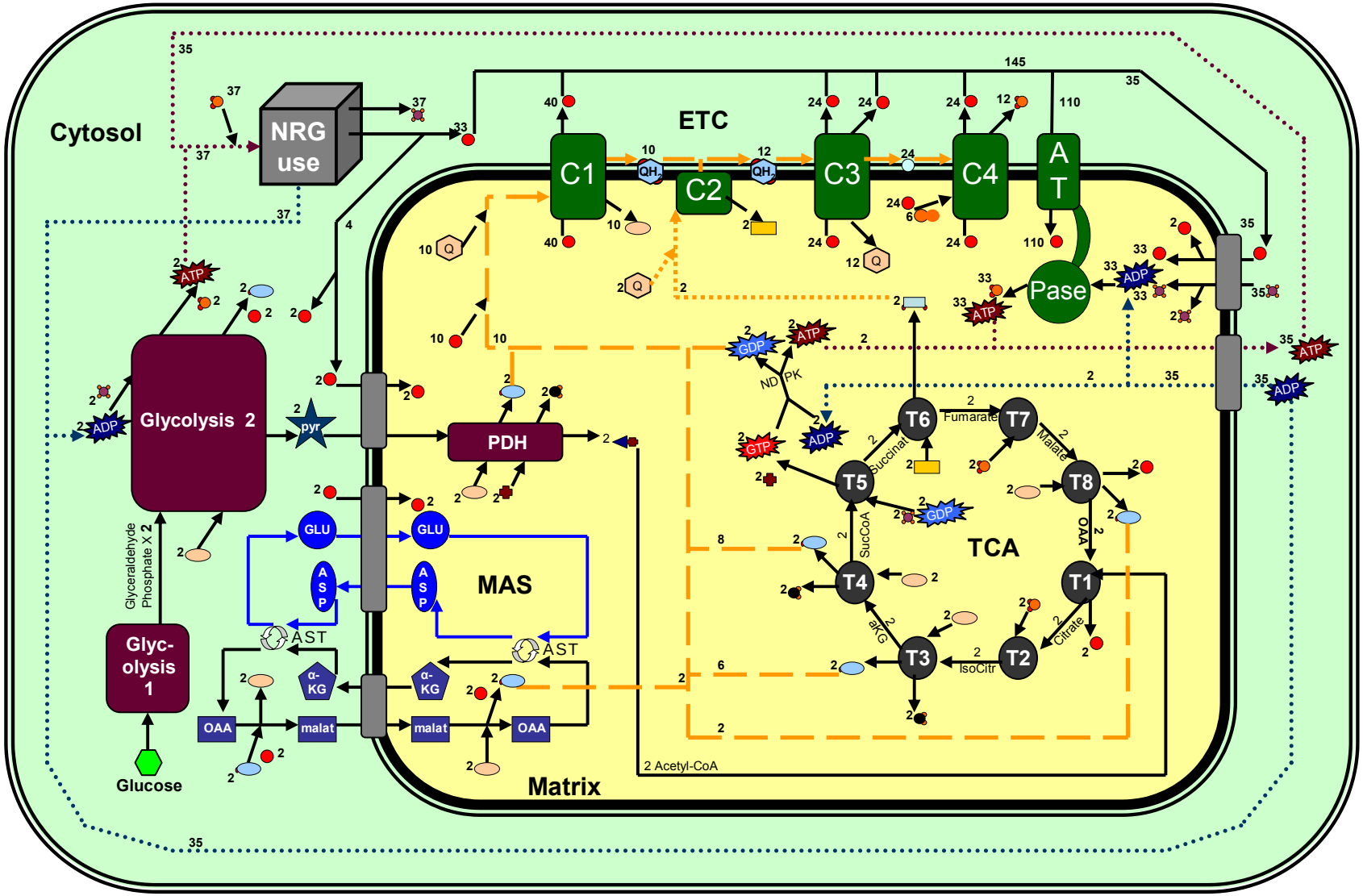


Figure 6. Comparison of Measured Literature P/O Ratio Ranges and Calculated Output Ranges by Substrate.



$3.33333\ (c)H^+ + 1\ (m)\ ADP + 1\ (m)\ P_i \rightarrow 1\ (m)\ ATP + 2.33333\ (m)\ H^+ + 1.\ (x)\ H_2O$

Figure 7. Summary of proposed ATP Production/Consumption Cycle.


Cytosol
NRG use
ETC
C1
C2
C3
C4
A T
Pase
Glycolysis 2
PDH
pyr.
GLU
ASP
AST
MAS
α-KG
OAA
malat
Glyc-olysis 1
Glucose
Glyceraldehyde Phosphate X 2
ND PK
T1
T2
T3
T4
T5
T6
T7
T8
TCA
Fumarate
Malate
Succinat
SucCoA
aKG
Citrate
IsoCitr
OAA
2 Acetyl-CoA
Matrix
ATP
ADP
GTP
GDP
Q
QH2

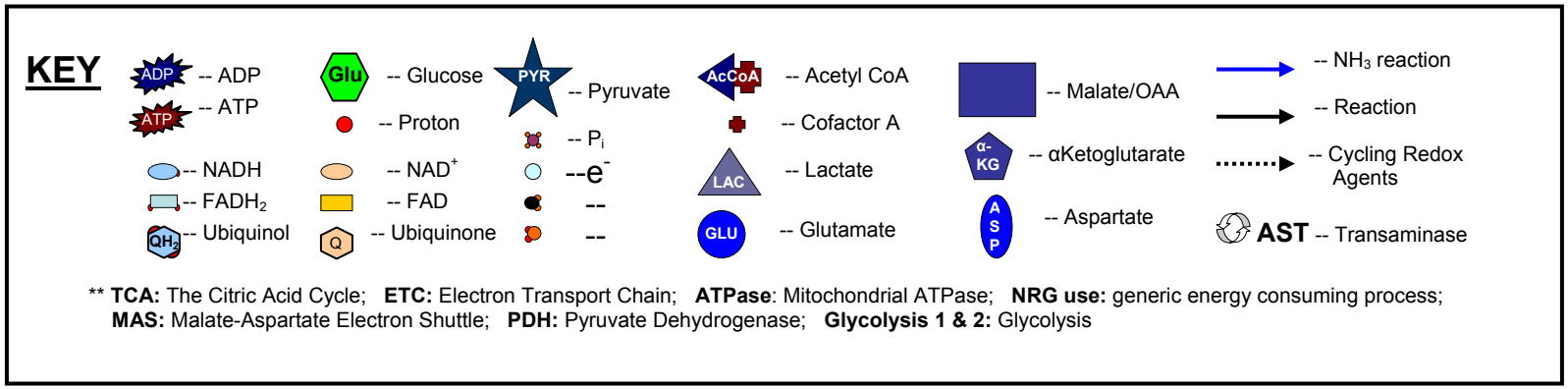


Figure 8. Schematic representation of the complete oxidative metabolism of glucose, utilizing the Malate-Aspartate electron shuttle, with an H^+/ATP ratio of 13/3. 37 molecules of ATP are generated and consumed by energy-dependent cytosolic processes.

Appendix I

The following is a compressed list of reactions and reaction summaries (wherever possible) used in the formulation of the modeled biochemical systems. The written equations are the direct output of a computer program that extracts the coefficients directly from the vectors internal to the program executing the linear algebraic manipulations that describe these systems.

Where possible, enzyme commission numbers (EC#s) have been provided so the reaction being referenced is unambiguous.

Many of the reactions listed are summaries (e.g. P99 – pyruvate dehydrogenase summary). In the case of P99, P11- P16 have also been included (the constituent reactions that make up P99) as a demonstration. Summary reactions are only used when their constituent reactions never produce intermediates that participate in side reactions. For example, all ETC complexes have complicated mechanisms involving many reactions, cytochromes, and non-redox reactants. However, in the systems studied, the redox intermediates never participate in any reactions outside of oxidative metabolism. This allows the series of reactions that always progress in a stoichiometrically identical way to be collapsed into a single, summarized reaction. This makes the system less cumbersome, easier to understand, and allows the program to run more quickly and efficiently.

System ID	Enzyme / Rxn Name	Compartment	EC #	BALANCED REACTION
G10	Hexokinase / Glucokinase	cytosol	2.7.1.1	1. (c)Adenosine-triphosphate + 1. (c)Glucose --> 1. (c)Adenosine-diphosphate + 1. (c)Glucose_6-phosphate + 1. (c)Proton
G11	Phosphoglucose isomerase	cytosol	5.3.1.9	1. (c)Glucose_6-phosphate --> 1. (c)Fructose_6-phosphate
G12	Phospho-fructokinase	cytosol	2.7.1.11	1. (c)Adenosine-triphosphate + 1. (c)Fructose_6-phosphate --> 1. (c)Fructose_1,6-bisphosphate + 1. (c)Adenosine-diphosphate + 1. (c)Proton
G13	Aldolase	cytosol	5.3.1.1	1. (c)Fructose_1,6-bisphosphate --> 1. (c)Glyceraldehyde_3-phosphate + 1. (c)Glycerone_phosphate
G14	Triosephospht Isomerase	cytosol	5.3.1.1	1. (c)Glycerone_phosphate --> 1. (c)Glyceraldehyde_3-phosphate
G15	G-3P dehydrogenase	cytosol	1.2.1.12	1. (c)Glyceraldehyde_3-phosphate + 1. (c)Nicotinamide_Adenine_dinucleotide_(ox) + 1. (c)Orthophosphate --> 1. (c)1,3-Bisphospho-glycerate + 1. (c)Proton + 1. (c)Nicotinamide_Adenine_dinucleotide_(red)
G16	Phospho-glycerate kinase	cytosol	2.7.2.3	1. (c)1,3-Bisphospho-glycerate + 1. (c)Adenosine-diphosphate --> 1. (c)Adenosine-triphosphate + 1. (c)3-Phospho-glycerate
G17	Phospho-glycerate mutase	cytosol	5.4.2.1	1. (c)3-Phospho-glycerate --> 1. (c)2-Phospho-glycerate
G18	Enolase	cytosol	4.2.1.11	1. (c)2-Phospho-glycerate --> 1. (c)Phosphoenolpyruvate + 1. (x)Water

G19	Pyruvate kinase	cytosol	2.7.1.40	1. (c)Phosphoenolpyruvate + 1. (c)Adenosine-diphosphate + 1. (c)Proton --> 1. (c)Adenosine-triphosphate + 1. (c)Pyruvate
cP50	Lactate dehydrogenase	cytosol	1.1.1.28	1. (c)Proton + 1. (c)Nicotinamide_Adenine_dinucleotide_(red) + 1. (c)Pyruvate --> 1. (c)Lactate + 1. (c)Nicotinamide_Adenine_dinucleotide_(ox)
P11	Pryuvate dehydration complex E1	matrix	1.1.1.28	1. (m)Carbanion_Thiamine_pyrophosphate + 1. (m)Proton + 1. (m)Pyruvate --> 1. (m)Addition_Compound
P12	Pryuvate dehydration complex E1	matrix	1.1.1.28	1. (m)Addition_Compound --> 1. (m)Hydroxyethel_Thiamine_pyrophosphate + 1. (x)Carbon_Dioxide
P13	Pryuvate dehydration complex E1	matrix	1.2.4.1	1. (m)Hydroxyethel_Thiamine_pyrophosphate + 1. (m)Lipoamide --> 1. (m)Acetyllipoamide + 1. (m)Carbanion_Thiamine_pyrophosphate
P15	Dihydrolipoyl transacetylase	matrix	1.2.4.1	1. (m)Acetyllipoamide + 1. (m)CoA --> 1. (m)Acetyl_CoA + 1. (m)Dihydrolipoamide
P16	Dihydrolipoyl dehydrogenase	matrix	1.2.4.1	1. (m)Dihydrolipoamide + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (m)Proton + 1. (m)Lipoamide + 1. (m)Nicotinamide_Adenine_dinucleotide_(red)
P99	Pyruvate Dehydration Complex--Summary	matrix	2.3.1.12	1. (m)CoA + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) + 1. (m)Pyruvate --> 1. (m)Acetyl_CoA + 1. (m)Nicotinamide_Adenine_dinucleotide_(red) + 1. (x)Carbon_Dioxide
T10	Citrate synthase	matrix	1.8.1.4	1. (m)Acetyl_CoA + 1. (m)Oxaloacetate + 1. (x)Water --> 1. (m)Citrate + 1. (m)CoA + 1. (m)Proton
T11	Aconidate hydratase	matrix	N/A	1. (m)Citrate --> 1. (m)cis-Aconitate + 1. (x)Water
T12	Aconidate hydratase	matrix	2.3.3.1	1. (m)cis-Aconitate + 1. (x)Water --> 1. (m)isocitrate
T13	Isocitrate dehydrogenase	matrix	4.2.1.3	1. (m)isocitrate + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (m)a-ketoglutarate + 1. (m)Nicotinamide_Adenine_dinucleotide_(red) + 1. (x)Carbon_Dioxide
T14	α-ketoglutarate dehydration complex	matrix	4.2.1.3	1. (m)CoA + 1. (m)a-ketoglutarate + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (m)Nicotinamide_Adenine_dinucleotide_(red) + 1. (m)Succinyl_CoA + 1. (x)Carbon_Dioxide
T15	Succinyl CoA synthetase	matrix	1.1.1.42	1. (m)Guanosine-diphosphate + 1. (m)Orthophosphate + 1. (m)Succinyl_CoA --> 1. (m)CoA + 1. (m)Guanosine-triphosphate + 1. (m)Succinate
T16	Succinate dehydrogenase	matrix	N/A	1. (m)Flavin_Adenine_dinucleotide_(ox) + 1. (m)Succinate --> 1. (m)Flavin_Adenine_dinucleotide_(red) + 1. (m)Fumarate
T17	Fumarase	matrix	6.2.1.4	1. (m)Fumarate + 1. (x)Water --> 1. (m)L-Malate

T18	Malate dehydrogenase	matrix	1.1.1.37	1. (m)L-Malate + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (m)Proton + 1. (m)Nicotinamide_Adenine_dinucleotide_(red) + 1. (m)Oxaloacetate
mT20	Malic Enzyme	matrix	1.1.1.38-40	1. (m)L-Malate + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (m)Nicotinamide_Adenine_dinucleotide_(red) + 1. (m)Pyruvate + 1. (x)Carbon_Dioxide
cT20	Malic Enzyme	cytosol	1.1.1.38-40	1. (c)L-Malate + 1. (c)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (c)Nicotinamide_Adenine_dinucleotide_(red) + 1. (c)Pyruvate + 1. (x)Carbon_Dioxide
O199	NADH-Q reductase (complex I)--summary	matrix	N/A	5. (m)Proton + 1. (m)Nicotinamide_Adenine_dinucleotide_(red) + 1. (m)Ubiquinone --> 4. (c)Proton + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) + 1. (m)Ubiquinol
O299	Succinate-Q reductase (complex II)--summary	matrix	N/A	1. (m)Flavin_Adenine_dinucleotide_(red) + 1. (m)Ubiquinone --> 1. (m)Flavin_Adenine_dinucleotide_(ox) + 1. (m)Ubiquinol
O399	Cytochrome reductase (complex III)--summary	matrix	N/A	2. (m)Cyt_c_(ox) + 1. (m)Ubiquinol --> 2. (c)Proton + 2. (m)Cyt_c_(red) + 1. (m)Ubiquinone
O499	Cytochrome oxidase (complex IV)--summary	matrix	N/A	4. (m)Cyt_c_(red) + 12. (m)Proton + 1. (x)Molecular_Oxygen --> 8. (c)Proton + 4. (m)Cyt_c_(ox) + 2. (x)Water
O50	ATP Synthase--summary	matrix	N/A	3.33333 (c)Proton + 1. (m)Adenosine-diphosphate + 1. (m)Orthophosphate --> 1. (m)Adenosine-triphosphate + 2.33333 (m)Proton + 1. (x)Water
mH12	Nucleoside Diphosphate Kinase	matrix	2.7.4.6	1. (m)Adenosine-diphosphate + 1. (m)Guanosine-triphosphate --> 1. (m)Adenosine-triphosphate + 1. (m)Guanosine-diphosphate
cX01	pyruvate carrier	Xport: c->m	None	1. (c)Proton + 1. (c)Pyruvate --> 1. (m)Proton + 1. (m)Pyruvate
cX02	MCT-1 (monocarboxylate transporter)	Xport: c->m	None	1. (c)Proton + 1. (c)Lactate --> 1. (m)Proton + 1. (m)Lactate
cX021	MCT-1 (monocarboxylate transporter)	Xport: c->m	None	1. (c)ß-HydroxyButyrate + 1. (c)Proton --> 1. (m)ß-HydroxyButyrate + 1. (m)Proton
cX022	MCT-1 (monocarboxylate transporter)	Xport: c->m	None	1. (c)Acetoacetate + 1. (c)Proton --> 1. (m)Acetoacetate + 1. (m)Proton
cX03	Dicarboxylate	Xport: c->m	None	1. (c)L-Malate + 1. (m)Orthophosphate --> 1. (c)Orthophosphate +

	transporter			1. (m)L-Malate
cX031	Dicarboxylate transporter	Xport: c->m	None	1. (c)Succinate + 1. (m)Orthophosphate --> 1. (c)Orthophosphate + 1. (m)Succinate
cX50	Orthophosphate carrier	Xport: c->m	None	1. (c)Proton + 1. (c)Orthophosphate --> 1. (m)Proton + 1. (m)Orthophosphate
mX51	ATP-ADP translocase	Xport: m->c	None	1. (c)Adenosine-diphosphate + 1. (m)Adenosine-triphosphate --> 1. (c)Adenosine-triphosphate + 1. (m)Adenosine-diphosphate
G3P99	Glycerophosphate Shuttle--Summary	Xport: c->m	N/A	1. (c)Proton + 1. (c)Nicotinamide_Adenine_dinucleotide_(red) + 1. (m)Flavin_Adenine_dinucleotide_(ox) --> 1. (c)Nicotinamide_Adenine_dinucleotide_(ox) + 1. (m)Flavin_Adenine_dinucleotide_(red)
MAS99	Malate Aspartate Shuttle--Summary	Xport: c->m	N/A	2. (c)Proton + 1. (c)Nicotinamide_Adenine_dinucleotide_(red) + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (c)Nicotinamide_Adenine_dinucleotide_(ox) + 2. (m)Proton + 1. (m)Nicotinamide_Adenine_dinucleotide_(red)
LAC99	Lactate Shuttle--Summary	Xport: c->m	N/A	2. (c)Proton + 1. (c)Nicotinamide_Adenine_dinucleotide_(red) + 1. (c)Pyruvate + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (c)Nicotinamide_Adenine_dinucleotide_(ox) + 2. (m)Proton + 1. (m)Nicotinamide_Adenine_dinucleotide_(red) + 1. (m)Pyruvate
NADHDH	NADH-DH--Summary	Xport: c->m	N/A	1. (c)Nicotinamide_Adenine_dinucleotide_(red) + 5. (m)Proton + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 4. (c)Proton + 1. (c)Nicotinamide_Adenine_dinucleotide_(ox) + 1. (m)Nicotinamide_Adenine_dinucleotide_(red)
HLK99	Proton Transport--Generic	Xport: c->m	N/A	1. (c)Proton --> 1. (m)Proton
H50	Energy Usage Function--Generic	matrix	N/A	1. (c)Adenosine-triphosphate + 1. (x)Water --> 1. (c)Adenosine-diphosphate + 1. (c)Proton + 1. (c)Orthophosphate
cK10	Betahydroxybutyrate dehydrogenase	cytosol	1.1.1.30	1. (c)ß-HydroxyButyrate + 1. (c)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (c)Acetoacetate + 1. (c)Proton + 1. (c)Nicotinamide_Adenine_dinucleotide_(red)
mK10	Betahydroxybutyrate dehydrogenase	matrix	1.1.1.30	1. (m)ß-HydroxyButyrate + 1. (m)Nicotinamide_Adenine_dinucleotide_(ox) --> 1. (m)Acetoacetate + 1. (m)Proton + 1. (m)Nicotinamide_Adenine_dinucleotide_(red)
mK20	Thiophorase	matrix	2.8.3.5	1. (m)Acetoacetate + 1. (m)Succinyl_CoA --> 1. (m)Aceto-acetyl_CoA + 1. (m)Succinate
mK21	Acetyl-CoA C-Acetyltransferase	matrix	2.3.1.9	1. (m)Aceto-acetyl_CoA + 1. (m)CoA --> 2. (m)Acetyl_CoA

Appendix II

This is the source code for the program that manipulates the mathematical representations of the biochemical systems studied here.

```
// Productions: RESEARCH
// Solving Matrices: Math utility to solve a 2D Matrix
// Author: Douglas Walled
// June 23rd, 2005


      /********************MATRIX SOLVER.cpp:  *************************
      **                                                               **
      **      This program will read a labelled 2D matrix in from a file. It will      **
      **      ask the user to specify which equations to consider when solving      **
      **      and to specify inhomogenous terms, if any. The template utilizes      **
      **      a third party numerical toolkit developed at the National Institute      **
      **      of Technology. The 2D Matrix represents a system of m molecules by **
      **      n biochemical reactions. A menu is provided to manipulate the      **
      **      inputs, retrieve outputs, and access SpecVBuilder.cpp and      **
      **      BalanceChecker.cpp                                      **
      **                                                               **
      *******************************************************************/


//*************************
//**    Header Files       **
//*************************
#include "DGW_jama_lu.h"    // Includes 3rd party TNT library
#include <fstream>          // Required for file I/O
#include <string>           // Need for use of 'string' class
#include <cstring>          // Required for strcmp()
#include <stdlib.h>

using namespace JAMA;       // TNT namespace
using namespace std;        // standard namespace
//#include <cstdlib>        // Required for strchr()


//*********************************
//**  Global Variables (reference only)  **
//*********************************
//const int nMolecules      // Number of Molecules in the System
//const int nReactions      // Number of Reactions in the System
//const double Produced     // Standard of Production/Consumption
//extern const int nAtoms   // Number of Atoms comprising system Molecules
//extern const int nSpecies // Number of Total species known to the model

extern int newsize;         // # Molecules after exclusion
extern int pref[nMolecules];    // Preferred order of molecules
extern char *spcfile;           // File containing molecular specifications
extern char *inspecies;         //File containing all Molecule definitions

char *infile = "Balanced Reactions.txt";  //File containing original matrix
char *outfile = "Summary.txt";          // File containing solution summary
char sTable[8];                         // Name of Reaction Matrix
char sMolecule[nMolecules][8];          // Array of Molecule Names
char sReactions[nReactions][8];         // Array of Reaction Names
extern char sAtoms[nAtoms][8];          // Array of Atom Names


extern Array2D<double> AllSpecies;          // Init Matrix for all molecule definitions

extern Array2D<double> SystemSpecies;       // Init Matrix for incld. molecule defs.
extern Array2D<double> NetAtoms;            // Init Matrix for net produced atomic species

Array2D<double>                             // Init Reaction Matrix object for program
      Reaction(nMolecules,nReactions);
Array2D<double>                             // Init Reaction Matrix object as an 'ORIGINAL'
      orgReaction(nMolecules,nReactions);
Array2D<double>                             // Init specification vector to 0 for all
      SpcV(nMolecules, 2, 0.0);
Array1D<double>                             // Solution x to the adjusted matrix
      Soln(nReactions, 0.0);
Array2D<double>                             // Overall production/destruction of species
      ProdVector(nMolecules,1, 0.0);
Array2D<double>                             // Hold molecular specifications from file
      MolecKey(nMolecules,2,0.0);


//*************************
//**  Function Prototypes      **
//*************************
int get_Reactions(char *);          // Reads Reaction Matrix from infile

char MainMenu();                    // Basic user interface
int display_Reactions();            // Prints modified Reaction Matrix to screen
int display_FileReact(char *);      // Prints Reaction Matrix from file to screen
int display_Pivots();               // Prints LU Objects' pivoting of Reaction Matrix
int check_Balance();                // Checks that equations being used are balanced
int display_ProdVect();             // Displays all species produced/destroyed
int reset_Matrix(char *);           // Resets Matrix to match original infile
int solve_System(void);             // Solves Matrix where possible and prints solution
```

```
int quick_Solve();                  // Automatically writes basic solution to file
int Summary(char *);                // Writes a summary of findings to a txt file
string EquationWriter();            // Writes the Equation

//Functions from SpcVBuilder file
int build_SpcV();                   // User specifies b, esp. which equations to include
int get_SpcV();                     // Displays current specifications
int file_Specs(char *);             // Writes molecule specs to SpcV from file
int perm_Matrix();                  // Permutes the reaction matrix according to SpcV



//**************************
//**   MAIN PROGRAM        **
//**************************
int main(void)
{
        int i;
        char choice;

        get_Reactions(infile);
        newsize = nMolecules;

        cout << "\nWELCOME TO MATRIX SOLVER V.2.0!!!\n";

  //initialize indices of SpcV from 0 to nMolecules
        for (i=0; i < SpcV.dim1(); i++) SpcV[i][0] = i;

        for(;;)
        {
                choice = MainMenu();

        try
        {
                switch(choice)
                {
                        case 'v': display_Reactions();
                                break;
                        case 'f': display_FileReact(infile);
                                break;
                        case 'p': display_Pivots();
                                break;
                        case 'c': check_Balance();
                                break;
                        case 's': build_SpcV();
                                break;
                        case 'o': solve_System();
                                break;
                        case 'u': display_ProdVect();
                                break;
                        case 't': reset_Matrix(infile);
                                break;
                        case 'x': quick_Solve();
                                break;
                        case 'm': Summary(outfile);
                                break;
                        case 'q':
                                return 0;       //Ends main()
                        default:
                                break;
                }
        }//END TRY
        catch(int i)
        {
                switch(i)
                {
                        case 1: cout << "\nMOLECULE NOT FOUND!!!\n\n";
                                break;
                        case 2: cout << "\nSAME SPECIFICATION!!! No change made\n\n";
                                break;
                        case 3: cout << "\nTRIVIAL CASE--SOLUTION IS 0!!!\n\n";
                                break;
                        case 4: cout << "\nPERMUTED MATRIX CAN'T BE SOLVED--CHANGE
                                        SPECIFICATIONS!\n\n";
                                break;
                        case 5: cout << "\nFILE NOT FOUND!!!\n\n";
                                break;
                        case 6: cout << "\nCANNOT BE SOLVED, MATRIX IS SINGULAR!!!\n\n";
                                break;
                        case 7: cout << "\nLESS EQUATIONS THAN VARIABLES! INCLUDE MORE
                                        MOLECULES!!!\n\n";
                                break;
                        case 8: cout << "\nONE OR MORE MOLECULES DOESN'T APPEAR IN "
                                        <<inspecies <<"!\n\n";
                                break;
                        default: cout << "\nUNDEFINED ERROR!!!\n";
                                break;
                }//END SWITCH
        }//END TRY
        }//END for(;;)
}//END main() -- Exits program



/*********** MainMenu(): Simple user interface prompts user to choose action***********
 ***********                and returns choice to execute a function above   **********/
char MainMenu()
{
        char ch;
        do
        {
                cout << "\n\n******Matrix Solver Main Menu (enter a letter):******\n\n";
                cout << "(v) View current matrix to be solved.\n";
                cout << "(f) View matrix from infile.\n";
                cout << "(p) View pivoting of matrix.\n";
                cout << "(c) Check to ensure that all reactions are truly balanced.\n";
```

```
                cout << "(s) Change specifications of system molecules.\n";
                cout << "(o) Solve current matrix.\n";
                cout << "(u) Displays all net consumption and production given current solution.\n";
                cout << "(t) Reset to current matrix to match file.\n";
                cout << "(x) QuickSolve it!\n";
                cout << "(m) Print current summary to file.\n";
                cout << "(q) Quit program.\n\n";

                cin >> ch;      cout << endl;

        }while(!strchr("vfpcsoutxmq",tolower(ch)));

        return tolower(ch);
}



/*********** get_Reactions(): Function reads Reaction matrix from file      ***********
 ***********                  and writes to respective global arrays        ***********/
int get_Reactions(char *)
{
        int i,j;

        ifstream in(infile, ios::in | ios::binary);

        if(!in)throw 5;


//      in >> nMolecules  >> nReactions;
        in >> sTable;

        for (j=0;j<nReactions;j++) in >> sReactions[j];

        for (i=0;i<nMolecules;i++)
        {
                in >> sMolecule[i];
                for (j=0;j<nReactions;j++)
                {
                        in >> Reaction[i][j];
                }
        }

        in.close();

//Makes "Backup copy" of Reaction matrix
        orgReaction = Reaction.copy();

        return 0;
}


/*********** display_Reactions(): Function prints Reaction matrix to        ***********
 ***********                      the screen.                               ***********/
int display_Reactions()
{
        int i,j;

        cout << "The current system of reactions is:\n\n";

        cout << sTable  << "\t";
        for (j=0;j<nReactions;j++) cout << sReactions[j] << "\t";
        cout << "\n";
        for (i=0; i < newsize ;i++)
        {
                cout << sMolecule[(int)SpcV[i][0]]  << "\t";
                for (j=0;j<nReactions;j++)
                {
                        cout << Reaction[i][j]  << "\t";
                }
                cout << "\n";
        }

        cout << endl << endl;

        cout << "Solving for the following b:\n";

        for(i=0; i<newsize; i++)
        {
                cout << "b[" << sMolecule[(int)SpcV[i][0]] << "]: "  << SpcV[i][1]  << "\n";
        }

        return 0;
}


/*********** display_FileReact(): Function prints Reaction matrix from      ***********
 ***********                      original file to the screen.              ***********/
int display_FileReact(char *)
{
        int i,j;

        cout << "The system of reactions is:\n\n";

        cout << sTable  << "\t";
        for (j=0;j<nReactions;j++) cout << sReactions[j] << "\t";
        cout << "\n";
        for (i=0; i < nMolecules ;i++)
        {
                cout << sMolecule[i]  << "\t";
                for (j=0;j<nReactions;j++)
                {
                        cout << orgReaction[i][j]  << "\t";
                }
                cout << "\n";
        }
```

```
	cout << endl << endl;

	return 0;
}



/*********** display_Pivots(): Function prints LU pivoting of original***********
 ***********                    matrix to the screen.                    ***********/
int display_Pivots()
{
	int i;

	Array2D<double> A = Reaction.copy();
	Array1D<int> P(Reaction.dim1());
	LU<double> B(A);
	P=B.getPivot();

	cout << "\nPivot vector P: \n";

	for (i=0; i < nMolecules; i++)
	{
		cout << "["  << i  << "]: "  << P[i]  << "\n";
	}
	cout << endl  << endl;

	return 0;
}



/*********** display_ProdVect(): Displays the production and use of all      ***********
 ***********                    species included in the system.              ***********/
int display_ProdVect()
{
	int i;

	Array2D<double> tmpSoln(nReactions,1, 0.0);
	for (i=0; i<nReactions; i++) tmpSoln[i][0] = Soln[i];

	ProdVector = matmult(orgReaction , tmpSoln);

	cout << "\n\nThe overall Production/Consumption vector is:\n";

	for (i=0; i < nMolecules ;i++)
	{
		cout << sMolecule[i]  << ":    \t" << ProdVector[i][0];
		cout << "\n";
	}
	return 0;
}
```

```
/*********** reset_Matrix(): Function restorest all values to match  ***********
 ***********                  those contained in the infile.          ***********/
int reset_Matrix(char *)
{
	int i;
	get_Reactions(infile);

	for(i=0; i < Soln.dim(); i++) {Soln[i] =0;}

	newsize = nMolecules;

  //reset SpecV
	for (i=0; i < SpcV.dim1(); i++) SpcV[i][0] = i;
	for (i=0; i < SpcV.dim1(); i++) SpcV[i][1] = 0;

	cout << "MATRIX SOLVER HAS BEEN FULLY RESET!";
	return 0;
}



/*********** Solve_System(): Function solves system of equations and       ***********
 ***********                  prints that solution                         ***********/
int solve_System(void)
{
	int i,j;
	int Rank = Reaction.dim2();
	Array2D<double> A(Rank,Rank);                    //Use only top 'RANK' equations
	Array2D<double> slvReaction(newsize,Rank); //Use first 'NEWSIZE' equations
	Array1D<double> x(Rank), b(Rank), InhomB(newsize);

  // Checks to see if enough molecules have been included
	if(newsize < Rank) throw 7;

	for (i=0;i<Rank;i++)
	{
		for (j=0;j<Rank;j++) A[i][j] = orgReaction[(int)SpcV[i][0]][j];
	}

	LU<double> B(A);

	//Solve and report solution if first 'RANK' equations can be solved without pivoting
	if (B.isNonsingular())
	{
		cout << "\nSystem of initial " << Rank << " preferred equations CAN be solved.\n";
		cout << "Using " << Rank << " out of " << newsize << " included molecules.\n";

		for(i=0; i < Rank; i++)  {b[i] = SpcV[i][1];}

		x = B.solve(b);
```

```
		for(i=0; i < Rank; i++) Soln[i] = x[i];

		cout << "\nThe solution is:\n";
		for(i=0; i < Soln.dim(); i++) {cout << "x[" << i << "]: " << Soln[i] << "\n";}

		return 1;
	}
	else cout << "\nSINGULAR system of first " << Rank << " equations CANNOT be solved.\n";


	//Allow for pivoting, but still do not allow excluded molecules.
	for(i=0; i < newsize; i++)  {InhomB[i] = SpcV[i][1];}
	for (i=0; i < newsize; i++)

	{
		for (j=0;j<Rank;j++) slvReaction[i][j] = orgReaction[(int)SpcV[i][0]][j];
	}

	LU<double> C(slvReaction);

	if (C.isNonsingular())
	{
		cout << "\nThe solution, utilizing all  " << newsize << " included molecules:\n";

		x = C.solve(InhomB);

		for(i=0; i < Rank; i++) Soln[i] = x[i];

		cout << "\nThe solution is:\n";
		for(i=0; i < Soln.dim(); i++) {cout << "x[" << i << "]: " << Soln[i] << "\n";}

		return 1;
	}
	else
	{
	ofstream out(outfile, ios::out | ios::trunc);
	if(!out)throw 5;

	out << "Most recent specifications led to no solution.";
	out.close();

	throw 6;
	}

	return 0;
}



/*********** Summary(char *): Function prints vital information     ***********
 ***********                   from current session to outfile       ***********/

int Summary(char *)
{
	int i,j;

	ofstream out(outfile, ios::out | ios::trunc);
	if(!out)throw 5;

	out << "SUMMARY FOR MATRIX SOLVER V 2.0:\n\n";

 // Print Overall Equation
	out << EquationWriter();
	out << "\n\n";

 // Print Critical Numbers for this run
	out << nMolecules << "\tnMolecules\n" << nReactions  << "\tnReactions\n";
	out << nAtoms << "\tnAtoms\n" << nSpecies  << "\tnSpecies\n";
	out << Produced << "\tProduced\n\n";

 // Print Molecules' names, specifications, and procuction/consumption
	//Better formatting if viewing summary in excel:
	out << "Molecule:\t\tSpecification:\t\tNet Produced:\n";
	//Better formatting if viewing summary as .txt file:
	//out << "Molecule:\tSpecification:\tNet Produced:\n";
	for(i=0; i < nMolecules; i++)
	{
		out << sMolecule[(int)SpcV[i][0]] << "\t\t";
	  //Better formatting if viewing summary in excel:
		if(SpcV[i][1] == -286.314159265359) out << "Excluded!" << "\t";
	  //Better formatting if viewing summary as .txt file:
	  //if(SpcV[i][1] == -286.314159265359) out << "Excluded!";
		else out << SpcV[i][1]  << "\t";

		out << "\t";

		out << ProdVector[(int)SpcV[i][0]][0] << "\n";
	}

 // Print Reaction Rates of Solution
	out << "\n\nReaction rates:\n";
	for(i=0; i < nReactions; i++)
	{
		out << sReactions[i] << "   \t";
		out << Soln[i]  << "\n";
	}

 // Print Original Matrix
	out << "\n\nOriginal Matrix read in from file:\n";
	// Labels
	out << sTable  << "\t";
	for (j=0;j<nReactions;j++) {out << sReactions[j]  << "\t";}
	out << "\n";
	// Molecules and coefficients in ORIGINAL order
	for (i=0;i<nMolecules;i++)
	{
```

```
                out << sMolecule[i]  << "\t";
                for (j=0;j<nReactions;j++)
                {
                        out << orgReaction[i][j]  << "\t";
                }
                out << "\n";
        }
        // Molecules and coefficients in PERMUTED order
/*      for (i=0;i<nMolecules;i++)
        {
                out << sMolecule[(int)SpcV[i][0]]  << "\t";
                for (j=0;j<nReactions;j++)
                {
                        out << orgReaction[(int)SpcV[i][0]][j]  << "\t";
                }
                out << "\n";
        }
*/

  // Print System Species definitions
        out << "\n\nIncluded Species are defined as follows:\n" << "Species"  << "\t";
        for (j=0; j<nMolecules; j++) out << sMolecule[j] << "\t";
        out << "\n";
        for (i=0; i < nAtoms ; i++)
        {
                out << sAtoms[i]  << "\t";
                for (j=0; j<nMolecules; j++)
                {
                        out << SystemSpecies[i][j]  << "\t";
                }
                out << "\n";
        }


  // Print NetAtoms production matrix to check balance
        out << "\n\nAtomic Balance Matrix is:\n" << "Atomic"  << "\t";
        for (j=0; j<nReactions; j++) out << sReactions[j] << "\t";
        out << "\n";
        for (i=0; i < nAtoms ; i++)
        {
                out << sAtoms[i]  << "\t";
                for (j=0; j<nReactions; j++)
                {
                        out << NetAtoms[i][j]  << "\t";
                }
                out << "\n";
        }

        out.close();

        cout << "\n\nSummary file successfully written!\n";

        return 0;
}



/*********** EquationWriter(): Function returns an equation that    ***********
 ***********                   represents current solution of system  ***********/
string EquationWriter()
{
        int i,j;
        double delta = .001;
        char number[10];
        string Equation;
        Equation = "";

        for (i=0; i < nMolecules ;i++)
                {
                  //Put net consumed (b < -0.001) species on left side of eqn
                        if(ProdVector[i][0] < -(delta))
                        {
                                _gcvt(ProdVector[i][0],9,number); //translates to char string

                          //Get rid of negative signs
                                for(j=0; number[j]; j++) {number[j] = number[j+1];}

                                Equation.insert(Equation.size(), number);   //writes coeff. to Equation

                                Equation += " ";

                                Equation += sMolecule[i];  //writes Species name to Equation

                                Equation += " + ";
        //DEBUG                 cout << Equation  << "\n";
                        }
                }
        Equation.erase(Equation.size()-3,2);    //Removes last "+" sign and one space

        Equation += " -->  ";

        for (i=0; i < nMolecules ;i++)
                {
                  //Put net produced (b > 0.001) species on right side of eqn
                        if(ProdVector[i][0] > delta)
                        {
                                _gcvt(ProdVector[i][0],9,number);           //translates to char string
                                Equation.insert(Equation.size(), number);   //writes coeff. to Equation

                                Equation += " ";

                                Equation += sMolecule[i];       //writes Species name to Equation
```

```
                        Equation += " + ";
        //DEBUG         cout << Equation  << "\n";
                    }
                }
        Equation.erase(Equation.size()-3,3);    //Removes last "+" sign and two spaces
        return Equation;
}
/*********** quick_Solve(): Function solves system of equations and          ***********
 ***********                 given infile preferences, prints to file          ***********/
int quick_Solve()
{
        int i;                          //Must initialize SpcV
        for (i=0; i < SpcV.dim1(); i++) SpcV[i][0] = i;

        file_Specs(spcfile);    // Load in user specified inclusion/inhomogenous values
        perm_Matrix();          // Permute Matrix according to user specifications
        solve_System();                 // Solve the current system
        display_ProdVect();     // Display net productions
        check_Balance();        // Checks to make sure original equations are balanced

        cout << "\nOverall System can be represented by the following equation:\n\n";
        cout << EquationWriter();

        Summary(outfile);       // Writes this solution to file called "Summary.txt"

        return 0;
}
```

```
// Productions: RESEARCH
// Specifying Solutions: Permutes Reaction Matrix for 'desirable solution'
// Author: Douglas Walled
// June 23rd, 2005


        /********************SpcVBuilder:  ************************************
        **                                                                    **
        **      This file contains the code necessary to provide a fully interfaced     **
        **      opportunity for the user to create a "specification vector" for a       **
        **      system of reactions being solved. This vector will be used to swap      **
        **      rows of the reaction matrix and present it to the LU decomposer so      **
        **      that molecules whose production/destruction rates are unknown are       **
        **      omitted from the square matrix being solved, and all other molecules    **
        **      have a specified rate of production/destruction (inhomogenous term).    **
        **                                                                    **
        ************************************************************************/

//*************************
//**    Header Files        **
//*************************
#include "DGW_jama_lu.h"    // Includes 3rd party TNT library
#include <fstream>          // Required for file I/O
#include <string>           // Need for use of 'string' class
using namespace JAMA;       // TNT namespace
using namespace std;        // standard namespace


//*************************
//**   Global Variables        **
//*************************
//extern const int nMolecules=10;     // Number of Molecules in the System
//extern const int nReactions=5;      // Number of Reactions in the System
//extern const double Produced=1;     // Standard of Production/Consumption
int newsize;                          // # Molecules after exclusion
int pref[nMolecules];                 // Preferred order of molecules
char *spcfile = "MolecKey.txt";       // File containing molecular specifications

extern char sTable[8];                          // Name of Reaction Matrix
extern char sMolecule[nMolecules][8];           // Array of Molecule Names
extern char orgMolecule[nMolecules][8];         // Array of 'Original' Molecule Names
extern char sReaction[nReactions][8];           // Array of Reaction Names
extern char orgReactions[nReactions][8];        // Array of 'Original' Reaction Names

extern Array2D<double> Reaction;                // Initialize Reaction Matrix object
```

```
extern Array2D<double> orgReaction;          // Initialize Reaction Matrix object 'ORIGINAL'
extern Array2D<double> SpcV;                 // Initialize specification vector to 0 for all
extern Array2D<double> MolecKey;             // Hold molecular specifications from file



//***************************
//**  Function Prototypes         **
//***************************
int build_SpcV();        // User specifies b, esp. which equations to include

char SpecMenu();         // Basic user interface
int get_Molecules();     // Displays list of molecules in the system
int get_SpcV();          // Displays current specifications
int file_Specs(char *);  // Writes molecule specs to SpcV from file
int change_SpcV();       // Alters SpcV through prompted user interface
int excclude_Mol();      // Excludes a molecule from being used in solution
int include_Mol();       // Reincludes a molecule for solution attempts
int      move_Mol();     // Move a molecule to the top of the current matrix
int perm_Matrix();       // Permutes the reaction matrix according to SpcV



/*************************************************************************
 *********** build_SpcV(): Function is as explained above. Due to its         ***********
 ***********                    size, it was put in a second file for clarity.       ***********
 *************************************************************************/
int build_SpcV()
{
        char choice;

        cout << "\nPLEASE SPECIFY MOLECULES FOR INCLUSION AND PRODUCTION RATES.\n";

        for(;;)
        {
                choice = SpecMenu();

        try
        {
                switch(choice)
                {
                        case 'm': get_Molecules();
                                break;
                        case 's': get_SpcV();
                                break;
                        case 'f': file_Specs(spcfile);
                                break;
                        case 'c': change_SpcV();
                                break;
                        case 'e': excclude_Mol();
                                break;
                        case 'i': include_Mol();
                                break;
                        case 'v': move_Mol();
                                break;
                        case 'p': perm_Matrix();
                                break;
                        case 'q':
                                return 0;          //Ends build_SpcV() call
                        default:
                                break;
                }
        }//END TRY
        catch(int i)
        {
                switch(i)
                {
                        case 1: cout << "\nMOLECULE NOT FOUND!!!\n\n";
                                break;
                        case 2: cout << "\nSAME SPECIFICATION!!! No change made.\n\n";
                                break;
                        case 3: cout << "\nTRIVIAL CASE--SOLUTION IS 0!!!.\n\n";
                                break;
                        case 4: cout << "\nPERMUTED MATRIX CAN NOT BE SOLVED -- CHANGE
                                        SPECIFICATIONS!!!\n\n";
                                break;
                        case 5: cout << "\nFILE NOT FOUND!!!\n\n";
                                break;
                        case 6: cout << "\nCANNOT MOVE AN EXCLUDED MOLECULE!!!\n\n";
                                break;
                        default: cout << "\nUNDEFINED ERROR!!!\n";
                                break;
                }//END SWITCH
        }//END TRY
        }//END for(;;)
}//END build_SpcV() -- return to call in 'switch(choice)' above



/*********** SpecMenu(): Simple user interface prompts user to choose action***********
 ***********                    and returns choice to execute a function above    ***********/
char SpecMenu()
{
        char ch;
        do
        {
                cout << "\n\n*******What would you like to do? (enter a letter)*******\n\n";
                cout << "(m) View list of molecules in the system\n";
                cout << "(s) View current specification values\n";
                cout << "(f) Set Molecule specifications from file.\n";
                cout << "(c) Change a molecule's specification\n";
                cout << "(e) Exclude a molecule from solution attempts\n";
                cout << "(i) Re-include a molecule for solution attempts\n";
                cout << "(v) Move a molecule to the top of the current matrix\n";
```

```
                cout << "(p) Permute Matrix according to current sepecifications\n";
                cout << "(q) Accept specifications and return to main menu\n\n";

                cin >> ch;      cout << endl;

        }while(!strchr("msfceivpq",tolower(ch)));

        return tolower(ch);
}



/*********** get_Molecules(): Displays numbered, ordered list of rows/      ***********
 ***********                        Molecules in Reaction Matrix.                    ***********/
int get_Molecules()
{
        int i;

        cout << "MOLECULES IN REACTION MATRIX(in order): \n";
        for(i=0; i < nMolecules; i++)
        {
                cout << i <<") " << sMolecule[i] << "  \t";
                if(!((i+1)%5)) cout << endl;
        }

        cout << endl  << endl;
        return 0;
}



/*********** get_SpcV(): Displays SpcV to the screen.       ***********
 ***********                                                  ***********/
int get_SpcV()
{
        int i,j;

        cout << "CURRENT MOLECULE SPECIFICATIONS: \n";
        for(i=0; i < nMolecules; i++)
                for(j=0; j<2; j++)
                {
                        if(!j) cout << "Molecule: " << sMolecule[(int)SpcV[i][0]] << "   \t";
                        if(j)
                        {
                                cout << "Spec: ";
                                if(SpcV[i][j] == -286.314159265359) cout << "Excluded!"  << endl;
                                else cout << SpcV[i][j]  << endl;
                        }
                }

        cout << endl;
        return 0;
}
```

```
/*********** file_Specs(): Reads file "Molec Key" and adjusts SpcV          ***********
 ***********                        accordingly.                                     ***********/
int file_Specs(char *)
{
        char Key[8];
        int i,j;

        cout << "Loading in specifications from file.\n\n";
        ifstream in(spcfile, ios::in | ios::binary);

        if(!in)throw 5;

        for(i=0; i<3; i++) in >> Key;

        for (i=0;i<nMolecules;i++)
        {
                in >> Key;
                in >> MolecKey[i][0];
                in >> MolecKey[i][1];
        }

        in.close();

// Automatically make appropriate changes to SpcV
        for (i=0;i<nMolecules;i++)
        {
                if(MolecKey[i][0] == 1)
                {
                        for(j=0; j<nMolecules; j++)
                        {
                                if(SpcV[j][0] == i)
                                SpcV[j][1] = -286.314159265359;
                        }
                }else for(j=0; j<nMolecules; j++)
                        {
                                if(SpcV[j][0] == i)
                                SpcV[j][1] = MolecKey[i][1];
                        }
        }
/*
//OUTPUTS FOR DEBUG
        cout << "MKey\tExcl\tSpc\n";  // Headers

        for (i=0; i < nMolecules; i++)  // Matrix
        {
                cout << sMolecule[i]  << "\t";
                cout << MolecKey[i][0]  << "\t"  << MolecKey[i][1]  << "\n";
        }
        cout << "\n\n";
        get_SpcV();             // SpcV out
```

```
*/
	return 0;
}
/*********** change_SpcV(): Prompts user through menu options to create***********
 ***********                        a "specification vector" for the reaction matrix***********/
int change_SpcV()
{
	int i,j;
	int index = -1;
	double s;
	string mol("");
	char indMolecule[nMolecules][8];	//lowercase index to check names against

	cout << "\nEnter the name of the molecule you would like to specify: \n";
	cout << "(Or type 'reset' to reset all to 0)\t";

	cin >> mol;				//User Inputs molecule name, converted to lowercase

	for(i=0; i < mol.length(); i++) mol[i] = tolower(mol[i]);

	if(mol == "reset")	//Resets if 'reset' was input and returns
	{
		for (i=0; i < SpcV.dim1(); i++) SpcV[i][1] = 0;
		cout << "\nAll specifications have been reset to 0(steady state).\n\n";
		return 0;
	}

  //Finds Molecule to be changed
	for(i=0; i < nMolecules; i++)
	{
		for(j=0; sMolecule[i][j]; j++) indMolecule[i][j] = tolower(sMolecule[i][j]);
		indMolecule[i][j] = '\0';
		if(indMolecule[i] == mol) index = i;
	}
	if(index<0) throw 1;	//Error handled if no match

  //Outputs current value, prompts for new value
	for(i=0; i < nMolecules; i++)
	{
		if(SpcV[i][0] == index)
		{
			cout << "\nCurrent specification of " << sMolecule[index] << " is "  <<
SpcV[i][1];
			cout << "\n\nSet new production/consumption value: \n";
			cout << "(X=0 for steady state, X<0 for consumed, X>0 for produced)  ";

			cin >> s;

			if(s == SpcV[i][1]) throw 2;	//Error if entry is same as current specification
			else SpcV[i][1] = s;

		  //Displays new value
			cout << "\nNew specification for " << mol << " is: ";
			if(SpcV[i][1] == -286.314159265359) cout << "Excluded!"  << endl;
			else cout << SpcV[i][1]  << endl;
		}
	}
	return 0;
}



/*********** exclude_Mol(): Prevents a molecdule from being considered   ***********
 ***********                        in the LU decomposition object.                  ***********/
int excclude_Mol()
{
	int i,j;
	int index = -1;
	string mol("");
	char indMolecule[nMolecules][8];	//lowercase index to check names against


	cout << "\nEnter the name of the molecule you would like to exclude: \n";

	cin >> mol;				//User Inputs molecule name, converted to lowercase
	for(i=0; i < mol.length(); i++) mol[i] = tolower(mol[i]);


  //Finds Molecule to be excluded, and does so.
	for(i=0; i < nMolecules; i++)
	{
		for(j=0; sMolecule[i][j]; j++) indMolecule[i][j] = tolower(sMolecule[i][j]);
		indMolecule[i][j] = '\0';
		if(mol == indMolecule[i]) index = i;
	}
	if(index<0) throw 1;			//Error handled if no match

	for(i=0; i < nMolecules; i++)
	{
		if(SpcV[i][0] == index) SpcV[i][1] = -286.314159265359;

	}
	cout << sMolecule[index] << " has been excluded.";

	return 0;
}



/*********** include_Mol(): Allows user to "put back" a molecule that   ***********
 ***********                        was once excluded from SpcV.                  ***********/
int include_Mol()
{
	int i,j;
	int index = -1;
	string mol("");
```

```
	char indMolecule[nMolecules][8];	//lowercase index to check names against


	cout << "\nEnter the name of the molecule you would like to Re-include: \n";

	cin >> mol;				//User Inputs molecule name, converted to lowercase
	for(i=0; i < mol.length(); i++) mol[i] = tolower(mol[i]);


  //Finds Molecule to be included, and does so.
	for(i=0; i < nMolecules; i++)
	{
		for(j=0; sMolecule[i][j]; j++) indMolecule[i][j] = tolower(sMolecule[i][j]);
		indMolecule[i][j] = '\0';
		if(mol == indMolecule[i]) index = i;
	}
	if(index<0) throw 1;	//Error handled if no match

	for(i=0; i < nMolecules; i++)
	{
		if(SpcV[i][0] == index)SpcV[i][1] = 0;
	}
	cout << sMolecule[index] << " is included, and has been set to 0.";

	return 0;
}



/************ move_Mol(): Allows user to move a molecule from anywhere  ************
 ************                    in SpcV to the top of SpcV. Others shift down. ************/
int	move_Mol()
{
	int i,j,marker;
	int index = -1;
	string mol("");
	char indMolecule[nMolecules][8];	//lowercase index to check names against

	cout << "\nEnter the name of the molecule you would like to move to the top: \n";

	cin >> mol;			//User Inputs molecule name, converted to lowercase
	for(i=0; i < mol.length(); i++) mol[i] = tolower(mol[i]);

  //Finds Molecule to be moved
	for(i=0; i < nMolecules; i++)
	{
		for(j=0; sMolecule[i][j]; j++) indMolecule[i][j] = tolower(sMolecule[i][j]);
		indMolecule[i][j] = '\0';
		if(mol == indMolecule[i])
		{
			index = i;		//Set index, then error if try to move excluded mol
			for(j=0; j<nMolecules; j++)
			{if(SpcV[j][0] == index && SpcV[j][1] == -286.314159265359)throw 6;}
		}
	}
	if(index<0) throw 1;	//Error handled if no match

  //Outputs for debug
//	cout << "\nPrevious order was: ";
//	for (j=0; j < newsize; j++) cout << sMolecule[pref[j]]  << " ";

  //Swap and permute
	for(j=1; j<newsize; j++)
	{
		if(pref[j] == index) marker = j;
	}

	for(j=marker; j >0; j--) {pref[j] = pref[j-1];}
	pref[0] = index;

//Outputs for debug
//	cout << "\n\nNew order is: ";
//	for (j=0; j < newsize; j++) cout << sMolecule[pref[j]]  << " ";

  //Copy SpcV into spv_tmp for storage
	Array2D<double> spv_tmp = SpcV.copy();

 //Permute SpcV
 //	cout << "\nSpcV: \n";
	for (i=0; i < newsize; i++)
	{
		SpcV[i][0] = pref[i];
		for(j=0; j < newsize; j++)
		{
			if(spv_tmp[j][0] == pref[i]) SpcV[i][1] = spv_tmp[j][1];
		}

//Outputs for Debug:
//		 cout << SpcV[i][0] << " " << SpcV[i][1] << "\n";

	  //Permute Reaction
		for (j=0; j<nReactions; j++)
			{
				Reaction[i][j] = orgReaction[pref[i]][j];
			}
	}
	return 0;
}



/************ perm_Matrix(): Permutes reaction matrix based on SpcV and  ************
 ************                 checks solvability. Permutes until solvable.     ************
 ************                 Prefers 1 net produced, and rest steady state. ************/
int perm_Matrix()
{
```

```
        int h, i, j, k, l, m;
        static int prod[nMolecules];
        static int std_st[nMolecules];
        static int excld[nMolecules];
        static int othr[nMolecules];


  //Finds and displays indices of molecules that are to be
  //excluded at steady state, or produced/consumed at known amount
        h=0;    j=0;     k=0;    l=0;     m=0;
        for(i=0; i < (nMolecules); i++)
        {
                if(abs(SpcV[i][1]) == Produced) prod[h++] = SpcV[i][0];
                else if(SpcV[i][1] == -286.314159265359) excld[j++] = SpcV[i][0];
                else if(SpcV[i][1] == 0) std_st[k++] = SpcV[i][0];
                else othr[m++] = SpcV[i][0];
        }
        excld[j] = '\0';  std_st[k] = '\0';          prod[h] = '\0';  othr[m] = '\0';
/* Outputs for debug
        cout << "\nstd_st: "; for(l=0; l<k; l++) cout << std_st[l] << " ";
        cout << "\nexcld: "; for(l=0; l<j; l++) cout << excld[l] << " ";
        cout << "\nprod: "; for(l=0; l<h; l++) cout << prod[l] << " ";
        cout << "\nother: "; for(l=0; l<m; l++) cout << othr[l] << " ";
*/
  //Permutes Reaction Matrix (and sMolecule) to suggest inclusion and exclusion
  //in solution. Will try to put things produced by 'Produced' near top,
  //then 'std_st', and 'excld' at bottom
        for(i=0; i<h; i++) pref[i] = prod[i];              //prod[el] first
        for(i=0; i<k; i++) pref[i+h] = std_st[i];          //std_st[el] next
        for(i=0; i<m; i++) pref[i+h+k] = othr[i];          //prod/cons next
        for(i=0; i<j; i++) pref[i+h+k+m] = excld[i];       //excld[el] at end
        newsize = nMolecules - j;                          //#Molecules after exclusion

/* Outputs for debug
        cout << "\n\nA preferred order is: ";
        for (i=0; i < nMolecules; i++) cout << pref[i]  << " ";
*/
        cout << "\nPERMUTING>>>>>>>>>>>>>\n\n";

  //Copy SpcV into spv_tmp for storage
         Array2D<double> spv_tmp = SpcV.copy();

  //Permute SpcV
  //      cout << "\nSpcV: \n";
        for (i=0; i < nMolecules; i++)
        {
                SpcV[i][0] = pref[i];
                for(j=0; j < nMolecules; j++)
                {
                        if(spv_tmp[j][0] == pref[i]) SpcV[i][1] = spv_tmp[j][1];
                }

   //             cout << SpcV[i][0] << " " << SpcV[i][1] << "\n";

          //Permute Reaction
                for (j=0; j<nReactions; j++)
                        {
                                Reaction[i][j] = orgReaction[pref[i]][j];
                        }
        }
        return 0;
}
// Productions: RESEARCH
// Solving Matrices: Math utility to solve a 2D Matrix
// Author: Douglas Walled
// August 8th, 2005


        /*******************BALANCE CHECKER.cpp:  **********************
        **                                                              **
        **      This file will read a labelled 2D matrix in from a file containing      **
        **      the atomic proportions of all molecules in the model. It will then      **
        **      build a matrix called SystemMolecules, which will contain ONLY the      **
        **      molecules being included in a particular run. It will then execute      **
        **      matrix multiplication with the balanced reaction matrix to check if     **
        **      the original reactions from file are in fact balanced. This is a        **
        **      secondary safegaurd intended to minimize human error, and is not        **
        **      essential to MatrixSolver.cpp's function.                               **
        **                                                              **
        ******************************************************************/


//**************************
//**    Header Files              **
//**************************
#include "DGW_jama_lu.h"    // Includes 3rd party TNT library
#include <iostream>
#include <fstream>          // Required for file I/O
#include <string>           // Need for use of 'string' class
#include <cstring>          // Required for strcmp()
using namespace JAMA;       // TNT namespace
using namespace std;        // standard namespace
//#include <cstdlib>        // Required for strchr()


//**************************
//**   Global Variables           **
//**************************
char *inspecies = "Species Library.txt";        //File containing all Molecule definitions
char *outspecies = "Balanced Molecules.txt";  //File with included Molecule definitions

extern int newsize;             // # Molecules after exclusion
extern int pref[nMolecules];    // Preferred order of molecules

extern char *spcfile;           // File containing molecular specifications
extern char *infile;            //File containing original matrix
extern char *outfile;           // File containing solution summary
```

```
extern char sTable[8];                  // Name of Reaction Matrix
extern char sMolecule[nMolecules][8];  // Array of Molecule Names
extern char sReactions[nReactions][8]; // Array of Reaction Names
char sAtoms[nAtoms][8];                 // Array of Atom Names

Array2D<double>                 // Init Matrix object containing all molecule definitions
        AllSpecies(nAtoms,nSpecies,77.0);
Array2D<double>                         // Init Matrix object containing incld. molecule defs.

        SystemSpecies(nAtoms,nMolecules,0.0);
Array2D<double>                         // Init Matrix to contain net produced atomic species
        NetAtoms(nAtoms,nReactions);
extern Array2D<double> Reaction;      // Init Reaction Matrix object for program to act on
extern Array2D<double> orgReaction;  // Init Reaction Matrix object as an 'ORIGINAL'
extern Array2D<double> SpcV;          // Init specification vector to 0 for all
extern Array2D<double> MolecKey;     // Hold molecular specifications from file



//***************************
//**  Function Prototypes          **
//***************************
int check_Balance();              // Checks that equations being used are balanced



/*************************************************************************
 ************ check_Balance(): Uses Matrix multiplication to check if          ************
 ************                      reactions are actually balanced.                ************
 *************************************************************************/
check_Balance()
{
        int i,j,k,l;
        string dmy(" ");                  // Dummy variable to waste input
//      float dum;                        // For C code trial
        int warning = 0;                  // Warning flag can be toggled
        int Found = 0;                    // Toggles whether molecule found or not
        double delta = 0.001;             // Error margin
        char indMolecule[nMolecules][8];        //lowercase index to check names against below:


        for(i=0; i<nMolecules; i++)
        {
                for(j=0; sMolecule[i][j]; j++) {indMolecule[i][j] = tolower(sMolecule[i][j+1]);}
        }


  // OPENS SpeciesLibrary.txt to read in the entire list of molecule definitions
        ifstream in(inspecies, ios::in | ios::binary);
        if(!in){cout << inspecies; throw 5;}

        in >> dmy;                        //Ignores table name

        for(i=0; i < nSpecies; i++)
        {
                in >> dmy;
        }

        for(k=0; k<nAtoms; k++)
        {
                in >> sAtoms[k];        // Loads in array of atom names
                for(i=0; i < nSpecies; i++)      // Constructs entire Species Library
                {
                        if(!in) in.open(inspecies);
                        in >> AllSpecies[k][i];
                }
        }
        in.close();


// OPENS SpeciesLibrary.txt again, writes table of Included Molecules, SystemSpecies
        ifstream in2(inspecies, ios::in | ios::binary);
        in2 >> dmy;                               // Ignores table name

        Found = 0;
        for(i=0; i < nSpecies ; i++)              // Searches first row
        {
                in2 >> dmy;

                for(l=0; l < dmy.length(); l++) dmy[l] = tolower(dmy[l]);

                for(j=0; j<nMolecules; j++)
                {
                        if(indMolecule[j] == dmy) // If finds included molecule, copies column
                        {
                                Found++;
                                for(k=0; k<nAtoms; k++) {SystemSpecies[k][j] = AllSpecies[k][i];}
                        }
                }
        }
        in2.close();


// Writes SystemMolecules to text file "Balanced Molecules.txt"
        ofstream out(outspecies, ios::out | ios::trunc);
        if(!out)throw 5;

        out << "Atomic"  << "\t";
        for (j=0; j<nMolecules; j++) out << sMolecule[j] << "\t";
        out << "\n";
        for (i=0; i < nAtoms ; i++)
        {
                out << sAtoms[i]  << "\t";
                for (j=0; j<nMolecules; j++)
                {
                        out << SystemSpecies[i][j]  << "\t";
```

```
                }
                out << "\n";
        }
        out.close();

        cout << "\n**"  << Found  << " species definitions have been included in SystemSpecies**";
        if(Found != nMolecules) throw 8;

 // Does SystemSpecies X orgReaction = NetAtoms
        NetAtoms = matmult(SystemSpecies, orgReaction);

 // Searches NetAtoms for nonzero entries, and reports molecules and reactions involved
        for(i=0; i<nAtoms; i++)
        {
                for(j=0; j<nReactions; j++)
                {
                        if(abs(NetAtoms[i][j]) > delta)
                        {
                                warning = 1;
                                cout << "\n\n******* WARNING!!! Unbalanced Reaction Found!!!
                                        *********\n";
                                cout << "Try checking the definitions of all molecules containing ";
                                cout << sAtoms[i] << " in reaction " << sReactions[j];
                                cout << "\nAlso check that all coefficients are balanced in the
                                        above reaction.";
                        }
                }
        }

 // Prints NetAtoms to screen if a warning is flagged, else provides pos. feedback
        if(warning)
        {
                cout << "\n\nThe NetAtoms Matrix is:\n\n" << "Atomic"  << "\t";
                for (j=0; j<nReactions; j++) cout << sReactions[j] << "\t";
                cout << "\n";
                for (i=0; i < nAtoms ; i++)
                {
                        cout << sAtoms[i]  << "\t";
                        for (j=0; j<nReactions; j++)
                        {
                                cout << NetAtoms[i][j]  << "\t";
                        }
                        cout << "\n";
                }
        }else cout << "\n*** ALL REACTIONS ARE BALANCED!!! ***\n";

        return 0;
}
```

The following files are the header files for the external dependencies of the above code. For the most part, these comprise part of a third party numerical toolkit developed at the National Institute of Technology (NIST).

DGW_Globals.h is a header file containing the system's global variables for easy access and alteration.

```
/**** DGW_Gloabals.h  *******
**                          **
**      Global variables.   **
**                          **
**************************/
//Global Variables of note!
const   int     nMolecules      =       xx      ;       // enter values for 'xx'
const   int     nReactions      =       xx      ;
const   int     nAtoms          =       xx      ;
const   int     nSpecies        =       xx      ;
const   double  Produced        =       xx      ;
```

```
/**** DGW_tnt.h  *******
**                      **
**  Include header files.  **
**                      **
**********************/

// Includes all relevant headers by only including dgw_tnt.h
#ifndef TNT_H
#define TNT_H
```

```
#include "dgw_Globals.h"              //Global Variables shared by program files

#include "tnt_array2d.h"              //2 Dimensional array class definition
#include "tnt_array2d_utils.h"        //2 Dimensional array class utils

#endif



/******** TNT_array2D.h   **********
**                                  **
**       Defining 2D Matrix type.   **
**                                  **
**********************************/

/*
*
* Template Numerical Toolkit (TNT): Two-dimensional numerical array
*
* Mathematical and Computational Sciences Division
* National Institute of Technology,
* Gaithersburg, MD USA
*
*
* This software was developed at the National Institute of Standards and
* Technology (NIST) by employees of the Federal Government in the course
* of their official duties. Pursuant to title 17 Section 105 of the
* United States Code, this software is not subject to copyright protection
* and is in the public domain. NIST assumes no responsibility whatsoever for
* its use by other parties, and makes no guarantees, expressed or implied,
* about its quality, reliability, or any other characteristic.
*
*/



#ifndef TNT_ARRAY2D_H
#define TNT_ARRAY2D_H

#include <cstdlib>
#include <iostream>
#ifdef TNT_BOUNDS_CHECK
#include <assert.h>
#endif

namespace TNT
{

/**
        Tempplated two-dimensional, numerical array which
        looks like a conventional C multiarray.
        Storage corresponds to C (row-major) ordering.
        Elements are accessed via A[i][j] notation.

        <p>
        Array assignment is by reference (i.e. shallow assignment).
        That is, B=A implies that the A and B point to the
        same array, so modifications to the elements of A
        will be reflected in B. If an independent copy
        is required, then B = A.copy() can be used.  Note
        that this facilitates returning arrays from functions
        without relying on compiler optimizations to eliminate
        extensive data copying.

        <p>
        The indexing and layout of this array object makes
        it compatible with C and C++ algorithms that utilize
        the familiar C[i][j] notation.  This includes numerous
        textbooks, such as Numercial Recipes, and various
        public domain codes.

        <p>
        This class employs its own garbage collection via
        the use of reference counts.  That is, whenever
        an internal array storage no longer has any references
        to it, it is destoryed.
*/
template <class T>
class Array2D
{


  private:
    T** v_;
        int m_;
    int n_;
    int *ref_count_;

        void initialize_(int m, int n);
    void copy_(T* p, const T*  q, int len) const;
    void set_(const T& val);
    void destroy_();
        inline const T* begin_() const;
        inline T* begin_();

  public:

    typedef        T   value_type;

                Array2D();
                Array2D(int m, int n);
                Array2D(int m, int n,  T *a);
                Array2D(int m, int n, const T &a);
    inline Array2D(const Array2D &A);
        inline Array2D & operator=(const T &a);
        inline Array2D & operator=(const Array2D &A);
```

```
        inline Array2D & ref(const Array2D &A);
              Array2D copy() const;
                    Array2D & inject(const Array2D & A);
        inline T* operator[](int i);
        inline const T* operator[](int i) const;
        inline int dim1() const;
        inline int dim2() const;
        inline int ref_count() const;
           ~Array2D();


};


/**
        Copy constructor. Array data is NOT copied, but shared.
        Thus, in Array2D B(A), subsequent changes to A will
        be reflected in B.  For an indepent copy of A, use
        Array2D B(A.copy()), or B = A.copy(), instead.
*/
template <class T>
Array2D<T>::Array2D(const Array2D<T> &A) : v_(A.v_), m_(A.m_),
        n_(A.n_), ref_count_(A.ref_count_)
{
        (*ref_count_)++;
}



/**
        Create a new (m x n) array, WIHOUT initializing array elements.
        To create an initialized array of constants, see Array2D(m,n,value).

        <p>
        This version avoids the O(m*n) initialization overhead and
        is used just before manual assignment.

        @param m the first (row) dimension of the new matrix.
        @param n the second (column) dimension of the new matrix.
*/
template <class T>
Array2D<T>::Array2D(int m, int n) : v_(0), m_(m), n_(n), ref_count_(0)
{
        initialize_(m,n);
        ref_count_ = new int;
        *ref_count_ = 1;
}



/**
        Create a new (m x n) array,  initializing array elements to
        constant specified by argument.  Most often used to
        create an array of zeros, as in A(m, n, 0.0).

        @param m the first (row) dimension of the new matrix.
        @param n the second (column) dimension of the new matrix.
        @param val the constant value to set all elements of the new array to.
*/
template <class T>
Array2D<T>::Array2D(int m, int n, const T &val) : v_(0), m_(m), n_(n) ,
        ref_count_(0)
{
        initialize_(m,n);
        set_(val);
        ref_count_ = new int;
        *ref_count_ = 1;

}

/**
        Create a new (m x n) array,  as a view of an existing one-dimensional
        array stored in <b>C order</b>, i.e. right-most dimension varying fastest.
        (Often referred to as "row-major" ordering.)
        Note that the storage for this pre-existing array will
        never be garbage collected by the Array2D class.

        @param m the first (row) dimension of the new matrix.
        @param n the second (column) dimension of the new matrix.
        @param a the one dimensional C array to use as data storage for
                the array.
*/
template <class T>
Array2D<T>::Array2D(int m, int n, T *a) : v_(0), m_(m), n_(n) ,
        ref_count_(0)
{
        T* p = a;
        v_ = new T*[m];
        for (int i=0; i<m; i++)
        {
                v_[i] = p;
                p += n;
        }
        ref_count_ = new int;
        *ref_count_ = 2;                /* this avoid destorying original data. */

}


/**
        Used for A[i][j] indexing.  The first [] operator returns
        a conventional pointer which can be dereferenced using the
        same [] notation.

        If TNT_BOUNDS_CHECK macro is define, the left-most index (row index)
        is checked that it falls within the array bounds (via the
```

```
        assert() macro.) Note that bounds checking can occur in
        the row dimension, but the not column, since
        this is just a C pointer.
*/
template <class T>
inline T* Array2D<T>::operator[](int i)
{
#ifdef TNT_BOUNDS_CHECK
        assert(i >= 0);
        assert(i < m_);
#endif

return v_[i];

}

template <class T>
inline const T* Array2D<T>::operator[](int i) const { return v_[i]; }

/**
        Assign all elemnts of A to a constant scalar.
*/
template <class T>
Array2D<T> & Array2D<T>::operator=(const T &a)
{
        set_(a);
        return *this;
}
/**
        Create a new of existing matrix.  Used in B = A.copy()
        or in the construction of B, e.g. Array2D B(A.copy()),
        to create a new array that does not share data.

*/
template <class T>
Array2D<T> Array2D<T>::copy() const
{
        Array2D A(m_, n_);
        copy_(A.begin_(), begin_(), m_*n_);

        return A;
}


/**
        Copy the elements to from one array to another, in place.
        That is B.inject(A), both A and B must conform (i.e. have
        identical row and column dimensions).

        This differs from B = A.copy() in that references to B
        before this assignment are also affected.  That is, if
        we have
        <pre>
        Array2D A(m,n);
        Array2D C(m,n);
        Array2D B(C);        // elements of B and C are shared.

</pre>
        then B.inject(A) affects both and C, while B=A.copy() creates
        a new array B which shares no data with C or A.

        @param A the array from elements will be copied
        @return an instance of the modifed array. That is, in B.inject(A),
        it returns B.  If A and B are not conformat, no modifications to
        B are made.

*/
template <class T>
Array2D<T> & Array2D<T>::inject(const Array2D &A)
{
        if (A.m_ == m_ &&  A.n_ == n_)
                copy_(begin_(), A.begin_(), m_*n_);

        return *this;
}




/**
        Create a reference (shallow assignment) to another existing array.
        In B.ref(A), B and A shared the same data and subsequent changes
        to the array elements of one will be reflected in the other.
        <p>
        This is what operator= calls, and B=A and B.ref(A) are equivalent
        operations.

        @return The new referenced array: in B.ref(A), it returns B.
*/
template <class T>
Array2D<T> & Array2D<T>::ref(const Array2D<T> &A)
{
        if (this != &A)
        {
                (*ref_count_) --;
                if ( *ref_count_ < 1 )
                {
                        destroy_();
                }

                m_ = A.m_;
                n_ = A.n_;
                v_ = A.v_;
                ref_count_ = A.ref_count_;
```

```
                (*ref_count_) ++ ;

        }
        return *this;
}

/**
        B = A is shorthand notation for B.ref(A).
*/
template <class T>
Array2D<T> & Array2D<T>::operator=(const Array2D<T> &A)
{
        return ref(A);
}

/**
        @return the size of the first dimension of the array, i.e.
        the number of rows.
*/
template <class T>
inline int Array2D<T>::dim1() const { return m_; }

/**
        @return the size of the second dimension of the array, i.e.
        the number of columns.
*/
template <class T>
inline int Array2D<T>::dim2() const { return n_; }


/**
        @return the number of arrays that share the same storage area
        as this one.  (Must be at least one.)
*/
template <class T>
inline int Array2D<T>::ref_count() const
{
        return *ref_count_;
}

template <class T>
Array2D<T>::~Array2D()
{
        (*ref_count_) --;

        if (*ref_count_ < 1)
                destroy_();
}

/* private internal functions */

template <class T>
void Array2D<T>::initialize_(int m, int n)
{


        T* p = new T[m*n];
        v_ = new T*[m];
        for (int i=0; i<m; i++)
        {
                v_[i] = p;
                p+=n;
        }
        m_ = m;
        n_ = n;
}

template <class T>
void Array2D<T>::set_(const T& a)
{
        T *begin = &v_[0][0];
        T *end = begin+ m_*n_;

        for (T* p=begin; p<end; p++)
                *p = a;

}

template <class T>
void Array2D<T>::copy_(T* p, const T* q, int len) const
{
        T *end = p + len;
        while (p<end )
                *p++ = *q++;

}

template <class T>
void Array2D<T>::destroy_()
{

        if (v_ != 0)
        {
                delete[] (v_[0]);
                delete[] (v_);
        }

        if (ref_count_ != 0)
                delete ref_count_;
}

/**
        @returns location of first element, i.e. A[0][0] (mutable).
*/
template <class T>
const T* Array2D<T>::begin_() const { return &(v_[0][0]); }
```

```
/**
        @returns location of first element, i.e. A[0][0] (mutable).
*/
template <class T>
T* Array2D<T>::begin_() { return &(v_[0][0]); }

/**
        Create a null (0x0) array.
*/
template <class T>
Array2D<T>::Array2D() : v_(0), m_(0), n_(0)
{
        ref_count_ = new int;
        *ref_count_ = 1;
}




} /* namespace TNT */

#endif
/* TNT_ARRAY2D_H */
```

```
/******* TNT_array2D_utils.h  ********
**                                   **
**       Tools for 2D Matrix type.   **
**                                   **
***************************************/
#ifndef TNT_ARRAY2D_UTILS_H
#define TNT_ARRAY2D_UTILS_H

#include <cstdlib>
#include <cassert>

namespace TNT
{


/**
        Write an array to a character outstream.  Output format is one that can
        be read back in via the in-stream operator: two integers
        denoting the array dimensions (m x n), followed by m
        lines of n  elements.

*/
template <class T>
std::ostream& operator<<(std::ostream &s, const Array2D<T> &A)
{
   int M=A.dim1();
   int N=A.dim2();

   s << M << " " << N << "\n";

   for (int i=0; i<M; i++)
   {
      for (int j=0; j<N; j++)
      {
         s << A[i][j] << " ";
      }
      s << "\n";
   }


   return s;
}

/**
        Read an array from a character stream.  Input format
```

```
        is two integers, denoting the dimensions (m x n), followed
        by m*n whitespace-separated elments in "row-major" order
        (i.e. right-most dimension varying fastest.)  Newlines
        are ignored.

        <p>
        Note: the array being read into references new memory
        storage. If the intent is to fill an existing conformant
        array, use <code> cin >> B;  A.inject(B) ); </code>
        instead or read the elements in one-a-time by hand.

        @param s the charater to read from (typically <code>std::in</code>)
        @param A the array to read into.
*/
template <class T>
std::istream& operator>>(std::istream &s, Array2D<T> &A)
{

    int M, N;

    s >> M >> N;

        Array2D<T> B(M,N);

    for (int i=0; i<M; i++)
       for (int j=0; j<N; j++)
       {
          s >>  B[i][j];
       }

        A = B;
    return s;
}


/**
        Matrix Multiply:  compute C = A*B, where C[i][j]
        is the dot-product of row i of A and column j of B.


        @param A an (m x n) array
        @param B an (n x k) array
        @return the (m x k) array A*B, or a null array (0x0)
                if the matrices are non-conformant (i.e. the number
                of columns of A are different than the number of rows of B.)


*/


template <class T>
Array2D<T> matmult(const Array2D<T> &A, const Array2D<T> &B)
{

        if (A.dim2() != B.dim1())
                return Array2D<T>();

        int M = A.dim1();
        int N = A.dim2();
        int K = B.dim2();

        Array2D<T> C(M,K);

        for (int i=0; i<M; i++)
                for (int j=0; j<K; j++)
                {
                        T sum = 0;

                        for (int k=0; k<N; k++)
                                sum += A[i][k] * B [k][j];

                        C[i][j] = sum;
                }

        return C;

}



} // namespace TNT

#endif
```

```
/********* DGW_jama_lu.h **********
**                                **
**    Code for LU Decomposition.  **
**                                **
*********************************/
#ifndef JAMA_LU_H
#define JAMA_LU_H

#include "DGW_tnt.h"

using namespace TNT;


namespace JAMA
{

  /** LU Decomposition.
  <P>
  For an m-by-n matrix A with m >= n, the LU decomposition is an m-by-n
  unit lower triangular matrix L, an n-by-n upper triangular matrix U,
  and a permutation vector piv of length m so that A(piv,:) = L*U.
  If m < n, then L is m-by-m and U is m-by-n.
  <P>
  The LU decompostion with pivoting always exists, even if the matrix is
  singular, so the constructor will never fail.  The primary use of the
  LU decomposition is in the solution of square systems of simultaneous
  linear equations.  This will fail if isNonsingular() returns false.
  */
template <class Real>
class LU
{
  /* Array for internal storage of decomposition.  */
  Array2D<Real>  LU_;
  int m, n, pivsign;
  Array1D<int> piv;


  Array2D<Real> permute_copy(const Array2D<Real> &A,
                             const Array1D<int> &piv, int j0, int j1)
      {
            int piv_length = piv.dim();

            Array2D<Real> X(piv_length, j1-j0+1);

    for (int i = 0; i < piv_length; i++)
      for (int j = j0; j <= j1; j++)
        X[i][j-j0] = A[piv[i]][j];

            return X;
      }

Array1D<Real> permute_copy(const Array1D<Real> &A,
            const Array1D<int> &piv)
      {
            int piv_length = piv.dim();
            if (piv_length != A.dim())
                  return Array1D<Real>();

            Array1D<Real> x(piv_length);


    for (int i = 0; i < piv_length; i++)
        x[i] = A[piv[i]];

            return x;
      }


      public :

/** LU Decomposition
@param  A   Rectangular matrix
@return     LU Decomposition object to access L, U and piv.
*/

 LU (const Array2D<Real> &A) : LU_(A.copy()), m(A.dim1()), n(A.dim2()),
            piv(A.dim1())

      {

// Use a "left-looking", dot-product, Crout/Doolittle algorithm.

        int i=0;
        int j=0;
        int k=0;

  for (i = 0; i < m; i++) {
    piv[i] = i;
  }
  pivsign = 1;
  Real *LUrowi = 0;;
  Array1D<Real> LUcolj(m);

  // Outer loop.

  for (j = 0; j < n; j++)
      {  // Make a copy of the j-th column to localize references.
```

```
        for (i = 0; i < m; i++)
                {
          LUcolj[i] = LU_[i][j];
        }

        // Apply previous transformations.

        for (int i = 0; i < m; i++)
                {
          LUrowi = LU_[i];

          // Most of the time is spent in the following dot product.

          int kmax = min(i,j);
          double s = 0.0;
          for (k = 0; k < kmax; k++)
                        {
            s += LUrowi[k]*LUcolj[k];
          }

          LUrowi[j] = LUcolj[i] -= s;
        }

        // Find pivot and exchange if necessary.

        int p = j;
//PKM         for (int i = j+1; i < m; i++) {
        for (i = j+1; i < m; i++)
                {
          if (abs(LUcolj[i]) > abs(LUcolj[p]))
                        {
                                p = i;
                        }
        }
        if (p != j)
                {
          for (k = 0; k < n; k++)
                        {
            double t = LU_[p][k];
                          LU_[p][k] = LU_[j][k];
                          LU_[j][k] = t;
          }
          k = piv[p];
                        piv[p] = piv[j];
                        piv[j] = k;
          pivsign = -pivsign;
        }

        // Compute multipliers.

        if ((j < m) && (LU_[j][j] != 0.0))
                {
          for (i = j+1; i < m; i++)
                                {
            LU_[i][j] /= LU_[j][j];
          }
        }
      }
    }


    /** Is the matrix nonsingular?
    @return     1 (true)  if upper triangular factor U (and hence A)
                                        is nonsingular, 0 otherwise.
    */

    int isNonsingular () {
      for (int j = 0; j < n; j++) {
        if (LU_[j][j] == 0)
          return 0;
      }
      return 1;
    }

    /** Return lower triangular factor
    @return     L
    */

    Array2D<Real> getL () {
            int nn= n<m ? n : m; //PKM
//PKM       Array2D<Real> L_(m,n);
      Array2D<Real> L_(m,nn);//PKM
      for (int i = 0; i < m; i++) {
//PKM           for (int j = 0; j < n; j++) {
        for (int j = 0; j < nn; j++) {//PKM
          if (i > j) {
            L_[i][j] = LU_[i][j];
          } else if (i == j) {
            L_[i][j] = 1.0;
          } else {
            L_[i][j] = 0.0;
          }
        }
      }
      return L_;
    }

    /** Return upper triangular factor
    @return     U portion of LU factorization.
    */

    Array2D<Real> getU () {
            int mm= n<m ? n : m; //PKM
//PKM               Array2D<Real> U_(n,n);
            Array2D<Real> U_(mm,n);//PKM
```

```
//PKM       for (int i = 0; i < n; i++) {
      for (int i = 0; i < mm; i++) {//PKM
        for (int j = 0; j < n; j++) {
          if (i <= j) {
            U_[i][j] = LU_[i][j];
          } else {
            U_[i][j] = 0.0;
          }
        }
      }
      return U_;
    }

    /** Return pivot permutation vector
    @return     piv
    */

    Array1D<int> getPivot () {
//PKM       return p;
      return piv;
    }


    /** Compute determinant using LU factors.
    @return     determinant of A, or 0 if A is not square.
    */

    Real det () {
      if (m != n) {
        return Real(0);
      }
      Real d = Real(pivsign);
      for (int j = 0; j < n; j++) {
        d *= LU_[j][j];
      }
      return d;
    }

    /** Solve A*X = B
    @param  B   A Matrix with as many rows as A and any number of columns.
    @return     X so that L*U*X = B(piv,:), if B is nonconformant, returns
                                          0x0 (null) array.
    */

    Array2D<Real> solve (const Array2D<Real> &B)
    {

            /* Dimensions: A is mxn, X is nxk, B is mxk */

      if (B.dim1() != m) {
                    return Array2D<Real>(0,0);
      }
      if (!isNonsingular()) {
        return Array2D<Real>(0,0);
      }

      // Copy right hand side with pivoting
      int nx = B.dim2();


            Array2D<Real> X = permute_copy(B, piv, 0, nx-1);

      // Solve L*Y = B(piv,:)
      for (int k = 0; k < n; k++) {
        for (int i = k+1; i < n; i++) {
          for (int j = 0; j < nx; j++) {
            X[i][j] -= X[k][j]*LU_[i][k];
          }
        }
      }
      // Solve U*X = Y;
      for (int k = n-1; k >= 0; k--) {
        for (int j = 0; j < nx; j++) {
          X[k][j] /= LU_[k][k];
        }
        for (int i = 0; i < k; i++) {
          for (int j = 0; j < nx; j++) {
            X[i][j] -= X[k][j]*LU_[i][k];
          }
        }
      }
      return X;
    }


    /** Solve A*x = b, where x and b are vectors of length equal
                to the number of rows in A.

    @param  b   a vector (Array1D> of length equal to the first dimension
                                                          of A.
    @return x a vector (Array1D> so that L*U*x = b(piv), if B is nonconformant,
                                          returns 0x0 (null) array.
    */

    Array1D<Real> solve (const Array1D<Real> &b)
    {

            /* Dimensions: A is mxn, X is nxk, B is mxk */

      if (b.dim1() != m) {
                    return Array1D<Real>();
      }
      if (!isNonsingular()) {
        return Array1D<Real>();
      }
```

```
            Array1D<Real> x = permute_copy(b, piv);

      // Solve L*Y = B(piv)
      for (int k = 0; k < n; k++) {
         for (int i = k+1; i < n; i++) {
              x[i] -= x[k]*LU_[i][k];
            }
         }

             // Solve U*X = Y;
//PKM       for (int k = n-1; k >= 0; k--) {
      for (k = n-1; k >= 0; k--) {
           x[k] /= LU_[k][k];
                    for (int i = 0; i < k; i++)
                    x[i] -= x[k]*LU_[i][k];
      }


      return x;
   }

}; /* class LU */

} /* namespace JAMA */

#endif
/* JAMA_LU_H */
```