Inverse Laplace Transformer

6.871 Final Project
Ashwin Deshpande
May 12th, 2005
Professor Davis

TABLE OF FIGURES

1 Abstract

The goal of the project was to design and implement a knowledge-based application system capable of being proficient in its domain. The Inverse Laplace Transformer (ILT) meets these specifications as an expert system in the field of symbolic inverse Laplace Transforms. The ILT performs this task by employing a customized rule-based system involving weight-based forward-propagation. While the ILT has many limitations, it contains the knowledge to perform elementary inverse Laplace transforms and the framework to easily expand into a more powerful system.

2 Introduction

The ILT was designed to act as an aid in performing inverse Laplace transforms. In addition to finding the inverse Laplace transform, the program provides meaningful steps describing the set of transformations it took throughout the process of ascertaining the inverse Laplace transform.

2.1 Design Parameters

As most real-life transfer functions are real and can be approximated as a rational function, the class of elementary, real, rational transfer functions has been chosen as the target input space, as this space is frequently used and does not require complex mathematics to reason about most inverse Laplace transforms in this space. Furthermore, the ILT's knowledge base is restricted to elementary operations that a college freshman could perform without the aid of calculus. However, while the knowledge base is confined, the general framework including expression storage and inference mechanisms is robust enough to incorporate additional knowledge and possibly extending the size of the program's input space.

The input to the program should be a well-formed scheme expression organized in a format described in Section 3. The output of the program should be a list of steps required to systematically derive the inverse Laplace transform from the given input.

2.2 Problem and Approach

$$F(s) = \int_0^{\infty} f(t)e^{-st}\,dt \qquad f(t) = \frac{1}{2\pi i}\int_{-\infty i}^{\infty i} F(s)e^{st}\,ds$$

Forward Laplace Transform Integral

Inverse Laplace Transform Integral

Figure 1 – Forward Laplace Transform and Inverse Laplace Transform Integrals

The Laplace transform is a well-defined formula. Furthermore, the inverse Laplace transform can also be written as a closed form integral expression as shown in figure 1. However, despite the ability to reduce all inverse Laplace transforms into integral evaluation, most humans rely on table properties to change the expression to a form that can be looked up in a table.

The ILT follows the human approach of using properties to simplify the s-Space expression into a table lookup. Furthermore, to emphasize the point that the system uses mainly properties, the ILT has been given only one table lookup to which it must simplify all s-Space expressions in order to transform into t-Space.

The ILT uses rules as the knowledge representation and a customized forward-chaining mechanism to propagate the rules through the rules. A rule-based approach was chosen as the problem-solving paradigm primarily as rules can closely mimic the human process of logically transforming one expression into another. Furthermore, a rule-based system is simple to both implement and expand upon.

3 Program Use

This section will describe the input and output formatting in section 3.1 and the show a demonstration of the program's output in section 3.2.

3.1 Program Formatting

The ILT is capable of receiving various input expressions and output various output expressions. A guide to the symbols used by the ILT is given in figure 2.

Symbol	Meaning
S	The main variable in Laplace space
T	The main variable in time space
E	The mathematical symbol e=2.718...
+	The addition operator
-	The subtraction operator
*	The multiplication operator
/	The division operator
G	The gamma function operator (Output Only)
P	The power operator
A+BI	A and B describe the real and imaginary components of a complex number

Figure 2 – Symbols and their meanings

The ILT is capable of receiving any expression involving the s, E, +, -, *, /, and P symbols and operators as well as any real numbers. The expressions must be well-formed scheme expressions where the operator is adjacent to the opening parenthesis to the left of it. Furthermore, after every number, symbol, operator, or close parenthesis, a space must separate the element from the next element.

The output of the program is similarly formatted. The final answer will be represented as a sum of t-Space expressions rather than a single t-Space expression. In addition, the output may also contain the gamma function in addition to the listed symbols and operators.

3.2 Program Demonstration

In this section, the ILT will perform an elementary inverse Laplace transform: F(s)= (/ 1 (P (+ s 5) 3)). The solution will be annotated to describe the output of the program.

The program first prompts the user for the file containing the input expression. The content of sampleexp3.txt is given in bold typeface.

Enter File Name: sampleexp3.txt
sampleexp3.txt: (/ 1 (P (+ s 5) 3))

Next, the program performs many rounds of forward propagation before reaching a solution. Only the solution path is displayed. The format of a solution entry is given in figure 3.

```
JUSTIFICATION
- Which Rule was Applied
        - A Description of the Rule
        - Which TransformPair the Rule was Applied on
        - The Local Expression Targeted

S-SPACE
        - The Sum of a Number of Expressions in s-Space
             -For each expression, a list of pending
              operations needed to undo the
              transforming steps in t-Space

T-SPACE
        - The Sum of a Number of Transformed
             Expressions in t-Space
```

Figure 3 – Format of solution

The first step converts the user input into a format that the program can use. Thus, no rules are applied at this point.

USER INPUT.
S-SPACE

```
      SUM {
            (/ 1+0I (P (+ s 5+0I ) 3+0I ) )
                  with pending operations:
      }

      T-SPACE
      SUM {
      }
      Press any key to continue . . .
```

The first rule applied on the solution path is an s-Space frequency shifting rule. The shift is applied to the previous transform to yield a new transform. Information about the target expression and rule description are also displayed.

```
Rule 'Frequency Shifting'
      Which performs: F(s-a) ==> F(s) with pending operation (+ a )
      Was applied on transform pair 0
      On the expression: (/ 1+0I (P (+ s 5+0I ) 3+0I ) )
      To yield the new transform:

      S-SPACE
      SUM {
            (/ 1+0I (P (+ (- s (+ 5+0I ) ) 5+0I ) 3+0I ) )
                  with pending operations: (Shift (+ 5+0I ) )
      }

      T-SPACE
      SUM {
      }
      Press any key to continue . . .
```

The next rule applied is a number evaluation rule. A few other rules similar in format to these examples are displayed; however, due to space considerations, they have been left out in this example.

```
      Rule 'Number Evaluator'
      Which performs: numerical expression => number
      Was applied on transform pair 0
      On the expression: (+ 5+0I )
      To yield the new transform:

      S-SPACE
      SUM {
            (/ 1+0I (P (+ (- s 5+0I ) 5+0I ) 3+0I ) )
                  with pending operations: (Shift (+ 5+0I ) )
      }

      T-SPACE
      SUM {
      }
      Press any key to continue . . .

      ……
```

When the s-Space expression is reduced to a form that the ‘Lookup’ rule can operate on, the expression is converted from s-Space to t-Space. Furthermore, all pending operations are applied on the t-Space expression.

```
Rule 'Lookup'
Which performs: (/ 1 (P s n)) --> (/ (P t (- n 1)) (G n))
Was applied on transform pair 0
On the expression: (/ 1+0I (P s 3+0I ) )
To yield the new transform:

S-SPACE
SUM {
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 5+0I ) ) ) ) (/ (P t (- 3+0I 1+0I )
) (G 3+0I ) ) )
}
Press any key to continue . . .
```

Finally, the program simplifies the t-Space expressions into a more compact form using rules and outputs the final answer.

```
Which simplifies to:
S-SPACE
SUM {
      DONE
}

T-SPACE
SUM {
      (/ (* (P E (* t -5+0I ) ) (P t 2+0I ) ) 2+0I )
}
```

A complete output sample highlighting many of the programs strengths can be found in appendix B.

4 Rules

The ILT uses rules as its primary knowledge representation. All the predicates in the system are transforms that are simplified or converted by the rules in order to make more transforms until a solution is reached. In this section, first the advantages and disadvantages of a rule-based system in this domain will be discussed. Next, the 3 different rule types will be outlined.

4.1 Rules as the Knowledge Representation

The ILT uses rules as the knowledge representation. This representation was chosen for various reasons.

Rules capture the human method of logically transforming one expression into another. Many humans understand a mathematical process best when expressions are systematically derived in succession from the starting expression to the goal expression. This characteristic of human understanding is best exemplified by mathematical proofs, which follow a very rigid logical structure. The transition from one expression to another can be well captured by rules, which are triggered by characteristics of the original expression and perform a logical operation to produce a new expression. Furthermore, the method of using rules enables the ILT to easily output its process after solving the inverse Laplace transform.

Rules are also simple to implement. This rule-based system has modularized rules, which can be easily added and subtracted without affecting the stability or independence of the inference engine. As this project was done on a tight schedule, the ease of implementing a rule-based system was a major factor in choosing rules as the knowledge representation.

4.2 Rule Types

The ILT uses three different types of rules. The three rule types are named always rules, lookup rules, and guess rules. These different types have varying costs and functions. Furthermore, these rules have different priorities, which will be described in detail in Section 5. A complete listing of all the rules is located in appendix A.

4.2.1 Always Rules

Always rules are the most primitive rules in the ILT. Most always rules implement simple arithmetic properties such as number evaluation, associativity, and commutability used for simplification. These rules generally target sub-expressions

rather than the entire transform. Always rules are used heavily to tweak expressions into forms on which the lookup and guess rules can operate. Furthermore, exclusively always rules are used in simplification. All always rules are designed to be non-conflicting, so the program will never enter an infinite loop by triggering only always rules. As their name suggests, always rules are always helpful in taking one transform and converting it to another simpler transform. Furthermore, always rules have no branching, as given an input transform, they will only produce at most one modified output transform. Since always rules nearly always simplify expressions and have low branching, they are assigned a very low cost.

Some examples of always rules are depicted in figure 4.

```
Number Evaluator: Cost 1.0f
     Reduces a numerical expression into a number
     Numerical expression => number

Multiplication Identity: Cost 1.0f
     Removes a factor 1 from a multiplication operation
     (* a 1 …  b ) => (* a … b)

Addition Simplifier: Cost 1.0f
     Merges nested addition operations
     (+ a (+ b … c ) d ) => (+ a b … c d)
```

Figure 4 – Always Rule Examples

4.2.2 Lookup Rule

The lookup rule is the only rule that can convert a transform from s-Space to t-Space. As described in section 2, the ILT has only one lookup rule to highlight the program's ability to use properties in finding inverse Laplace transforms. Since the lookup rule is the only one rule capable of bridging the conversion from s-Space to t-Space, it is used in every inverse Laplace transform. Furthermore, since the use of the lookup rule signals that at least part of the problem has been solved, the lookup rule is given no cost. In contrast, the more instances that transforms use the lookup rule, the more they are rewarded by subtracting penalties from their total cost. Thus, the application of the lookup rule is highly encouraged by the system.

The lookup rule is shown in figure 5.

```
Lookup: Cost 0.0f
     Transforms an expression from s-Space to t-Space
     (/ 1 (P s n)) --> (/ (P t (- n 1)) (G n))
```

Figure 5 – The Lookup Rule

4.2.3 Guess Rules

Guess rules, as their name suggest, guess that a particular complex operation is needed to perform the inverse Laplace transform. Some guess rules target the entire transform, whereas others only target sub-expressions within the transform. Furthermore, some guess rules permanently change the value of the s-Space expression by adding pending operations to the transform. Rules in this class include complex rules like generalized partial fractions, quadratic factoring, and applying linearity. Most transforms generated by guess rules are not used; however, a small subset of these applications create simplified transforms on which the system can then successfully operate. Since guess rules create many branches, are generally more complex than the other rules, and are not extensively used on the solution path, guess rules incur high costs.

Some examples of guess rules are listed in figure 6.

```
Scaling: Cost 200.0f
     Performs frequency scaling and adds a
          corresponding pending operation
     F(as) => F(s) with pending operation (SCALE a)

Quadratic Factorization: Cost 200.0f
     Performs quadratic factorization on an unfactored
          second degree polynomial
     (as^2 + bs + c) => (s + r1)(s + r2)

Subtraction Addition Conversion: 350.0f
     Converts a subtraction operation into an addition
          operation
     (- a b ) => (+ a (* -1 b ) )
```

Figure 6 – Guess Rule Examples

5 Architecture

This section will describe the architecture of the ILT system. The architecture of ILT was inspired by SAINT, and many elements of the architectures have similarities. The first section will describe the costs associated with transforms for forward-propagation. The second section will describe the simplification architecture. Finally, the third section will describe the entire system including the forward-propagation mechanism.

5.1 Costs

Every transform in the ILT system has a cost associated with it. Costs can be decomposed into complexity costs and rule costs. The lower the cost, the more

appealing the transform is to the system. The total cost for a transform is a weighted sum of the complexity cost and the rule cost.

5.1.1 Complexity Cost

The complexity cost is a rough measure of how difficult an expression is to transform. The complexity cost is calculated in a similar manner as SAINT's complexity function. The complexity is simply calculated as the summed nesting of all sub-expressions within the larger expression.

```
(+ (/ 1 s ) 2 )
```

Figure 7 – A Sample Expression for Complexity Cost Analysis

So, for example, if the given expression is of the form given in figure 7, the total complexity cost would be 6 as the 1 and s elements are contained within 2 levels of nesting and the / and 2 elements are contained within 1 level of nesting.

This way of measuring complexity is very inaccurate and could be a possible source of causing unnecessary computation. However, this crude approximation is sufficient to assign prohibitively high costs to extremely complex expressions that do not have high chances of yielding a simple transform.

5.1.2 Rule Cost

The rule cost is a measure of high much work has been performed on a given transform. If significant work has gone into a transform, yet the transform has not simplified; then, the rule costs will discourage the system from pursuing this transform as there probably are other transforms that can simplify more easily. The rule cost for a transform is calculated as the sum of the costs of all the rules applied to the input transform to reach the current transform.

There are exceptions to the calculation of rule costs. Performing quadratic factoring and especially partial fractions can easily lead to very large and complex expressions that have prohibitively high complexity costs, even though the transform

might be useful. To get around this problem the result of the quadratic factoring and partial fractions rules are simplified using the simplifier described in section 5.2. Furthermore, as the simplification would incur steep rule costs as a result of many elementary rules operating on the expressions, the simplified transforms of these two rules do not accrue rule costs while simplifying.

Rule costs serve to help simplify the complexity and number of steps of the final solution.

5.2 Simplification Propagation

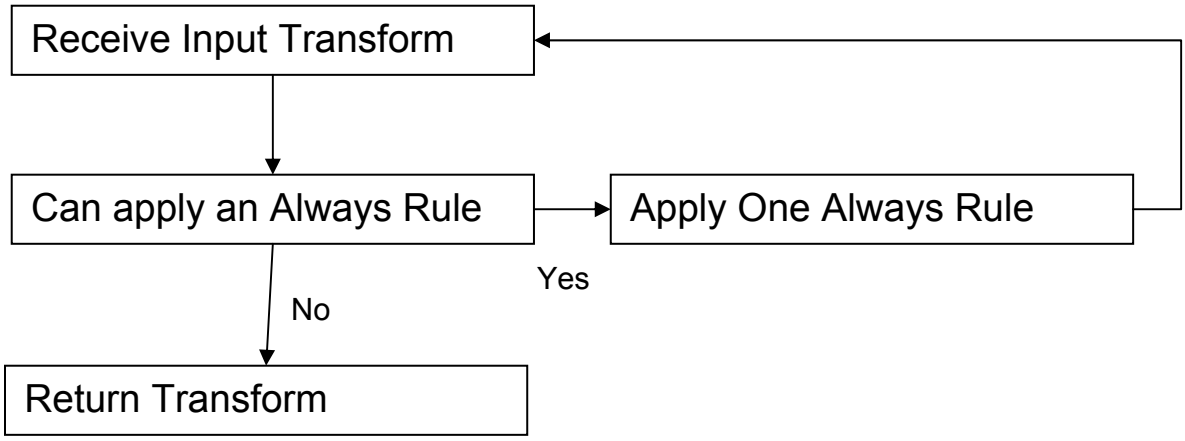

Figure 8 – Simplification Architecture

The ILT performs pure simplification steps when performing quadratic factoring, partial fractions, and t-Space simplification at the very end. The ILT performs this step by successively applying a single always rule to derive a simpler expression until no more always rules apply as shown in figure 8. Always rules are applied continuously regardless of the cost of the current transform.

5.3 Inverse Laplace Transform Propagation

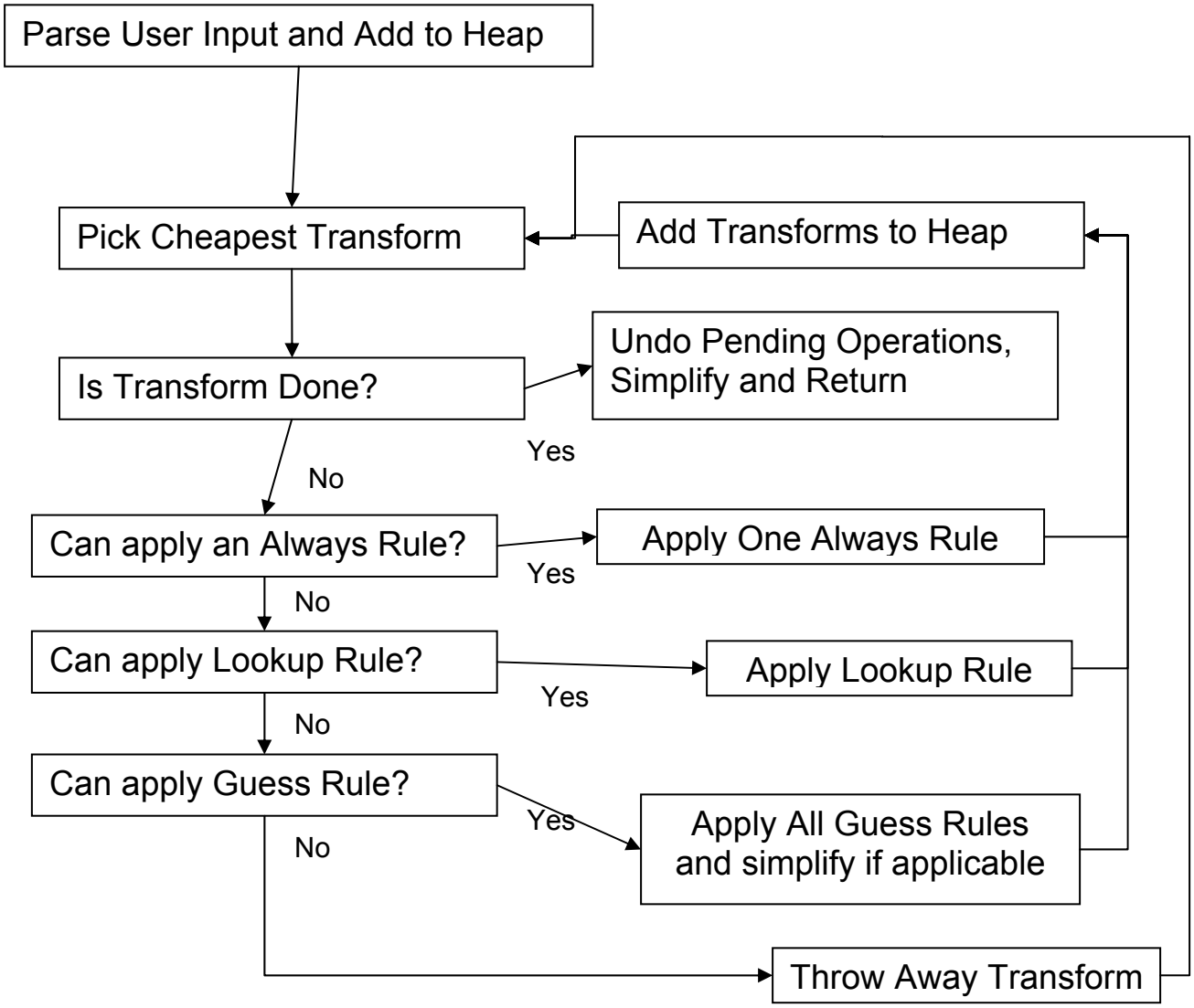

Figure 9 – Inverse Laplace Transform Propagation Architecture

The ILT uses a custom weight-based forward-chaining propagation mechanism as its inference engine as shown in figure 9. The program starts by

parsing the user input and adding the input transform to a heap. The rule cost of the initial transform is set to 0 as no rules have yet been applied.

Then the program enters a cycle of generating new transform predicates by applying rules to promising transforms. The most promising transform is the one with the lowest total score as described in section 5.1.

Next, the chosen transform is checked to see if it is done. If all s-Space expressions have been transformed, then the pending operations are performed on the t-Space expressions. Finally, those expressions are simplified using the simplification steps described in section 5.2, and the resulting transform is returned as the solution. If the transform is not finished, then the rules attempt to operate on it.

First, the transform is given exclusively to the always rules. If any always rule is triggered by the transform, only that rule is applied once. Then the original transform is removed from the heap and the new transform is inserted. The process is then restarted.

If no always rules are applicable, the lookup rule checks if any parts of the transform can be transformed to t-Space. If the lookup rule is triggered, a similar sequence of steps is taken as the original expression is discarded and the new expression is added.

If the lookup rule does not apply, then the guess rules are called. Multiple guess rules are allowed to trigger and fire on one propagation step, since the most appropriate guess rule is not known beforehand. Furthermore, if a single guess rule triggers multiple times, then each instance of the rule triggering will lead to a new modified transform. The original transform is discarded and all the new transforms are added to the heap. If no guess rules apply, then the system simply removes the original transform from the heap.

The propagation process keeps repeating until the problem is solved, the heap is empty (at which point the system does not know how to find the inverse Laplace transform), or the system runs out of resources. Due to the cyclic nature of the forward-propagation mechanism, some transforms may take a long time to return a solution and others may propagate infinitely.

6 Program Review

This section will provide a review of the program as a whole and of its various components. Both positive and negative aspects of the program will be discussed. Section 6.3 will describe possible future improvements to the program.

6.1 Positive Aspects

6.1.1 Successful Knowledge Representation

Rules as the knowledge representation proved to be a major success as rules easily capture the knowledge of logically deriving one transform from another. The rule-based system allows for modularity between the rules and the inference engine. Furthermore, the program applies rules in a similar format as human thought; so, describing the process of solving an inverse Laplace transform is easily conveyed to the user via the rules. Using a rule-based approach was a major factor in the success of this program.

6.1.2 System Generality

One positive aspect of the ILT system is the generality of input problems and the manner in which the program solves them. The ILT is not limited to a finite set of inputs. Rather, the ILT can attempt to solve any regular s-Space expression. This generality makes the ILT powerful as the system can generate arbitrarily large search trees on many classes of equations and can discover transforms that it was not even designed to solve. This generality also enables the ILT to never become outdated as one can never exhaust the full power of the system as there will always be new problems to solve that the ILT can handle.

6.1.3 System Scalability

Another positive aspect of the ILT is the framework in which it was designed. The forward-propagation mechanism is a robust system that can easily handle the addition of appropriately constructed rules.

At this point in time, the ILT cannot successfully transform all rational functions as its goal states. This is due to the lack of rules to handle certain types expressions, even within the rational function domain.

```
     (/ 1 (/ 1 (/ 1 s ) ) )
     Fails as the program does not know about nested
division.

     (/ 1 (+ (* (+ s 1 ) (- s 1 ) ) 1 ) )
     Fails as the program does not know how to expand
multiplication.

     (/ 1 (/ 1 (+ (/ 1 s ) (/ 1 (+ s 1 ) ) ) ) )
     Fails as the program does not know how to combine
fractions.
```

Figure 10 – Rational Functions for which ILT cannot find the inverse Laplace Transform

---

Figure 10 details three expressions that the current system cannot handle. However, the robustness of the inference engine allows the addition of rules to handle these cases. Already, there are class prototypes for these exceptions named 'DoubleDiv,' 'Expansion,' and 'AddFractions' respectively; however, these rules have not yet been implemented. Regardless, the addition of these rules poses no issues as to the stability or complexity of the entire system due to the scalable propagation mechanisms.

Furthermore, the power of the system can also easily be increased by adding more lookup transforms and other rules that enable the system to not have to rely on only first principles. In addition, the system can also support non-rational function transforms by adding the appropriate lookups and property rules. One simple example of extending the system to handle non-rational functions is the addition of a time-shifting rule that corresponds to exponential multiplication in the frequency

domain. Such a rule can easily be engineered using pending operations and the flexible propagation framework.

Thus, one of the most appealing elements of the ILT system is its scalability and its robust framework.

6.2 Negative Aspects

6.2.1 Halting Problems

One major limitation of the ILT system is its inability to determine if it can solve a particular transform or not. There exist many transforms, which the ILT will continuously expand into larger and more complex transforms in the hopes of the equation collapsing. However, if the ILT does not have the necessary tools, the transforms may continue to grow infinitely or loop.

One solution to this problem is to let the ILT time out after a set period of time when it is very unlikely that the system will find a solution. However, this approach has the obvious drawback that limiting the time given to the ILT system limits the range of transforms that the system can successfully operate on.

This issue may also be a consequence of the problem domain, as mathematicians have discovered integrals and consequently inverse Laplace transforms that cannot be evaluated in a closed form expression. It may also be the case that this problem is not Turing-decidable. Thus, there may be no remedy to completely solve this issue.

6.2.2 Resource Problems

Another issue of the ILT is the amount of resources that the program requires in order to run successfully. On difficult inputs, the ILT can consume over 100MB of RAM. While the system was not designed with efficiency and optimization in the foreground, the resource use of the system is a major limitation. Memory issues are the main reason the problem can handle 3rd degree denominators but cannot handle 4th degree denominators.

One possible way to improve memory usage is to reduce the branching factor. Currently, the ILT may continue to expand two identical transforms if they have different histories. Furthermore, the program may perform the same operations on identical transform pairs across many different transforms. One way to reduce this branching is to propagate transform pairs instead of entire transforms and check for redundancy on every propagation.

However, despite these solutions, the nature of the problem itself demands at least some amount of exponential branching in order to reach a solution. Thus, while memory issues can be suppressed, they cannot be completely eliminated.

6.3 Future Improvements

6.3.1 Caching/Case-Based Reasoning

One wasteful aspect of the ILT system is its inability to learn from examples. A way to improve the performance of the ILT system is to cache all previously successfully transformed expressions and always compare if an expression has been transformed previously before attempting to find the inverse Laplace transform through further forward-propagation. This method will allow the program to reuse previous work and avoid redundant computation.

An even greater improvement over caching is using a case-based reasoning mechanism in addition to forward propagation. Thus, if the system ever detects that it has performed a similar transform in the past; it can reweight the rule functions to favor the path of the successful transform as similar solution strategies will probably be applicable in the case of the new transform. With this capability, the propagation mechanism can receive hints about which rules might be more successful. Thus, the program would have a higher overall successful rule hit rate.

6.3.2 Cost Variations

Another change that can dramatically affect the performance of the ILT is the cost function calculations. The current complexity cost calculation is quite arbitrary

in some cases. A refined heuristic can probably be developed which can evaluate the transform complexity more accurately. Given such a heuristic, the program would pick less complex solutions more often, leading to a higher overall successful rate.

Another possible area for improvement involving costs is weighting of different rules. To arrive at the current weights for the rules, all rules were given arbitrary costs and were tweaked by hand until the program responded well enough. However, using a machine learning approach such as neural nets or SVMs, the system can be given assorted transforms and a more optimal set of weightings can be computed. At the present, the efficiency improvement from such an effort is unknown; however, variations in rule costs have produced radically different responses from the system during testing.

7 Lessons Learned

This section will outline some of the lessons learned from designing and implementing the ILT system. The major lessons learned were regarding completeness of systems and a comparison of pre-existing versus customized control structures.

7.1 Completeness Issues

In designing the system, I became aware of how difficult it is to create a complete transformer capable of handling every transform in a certain class, even for elementary classes such as the class of rational transforms. The major issue I faced was the enumeration of every necessary rule needed to enable the program to handle any transform. The method I used to generate a set of rules was testing the system on assorted types of transforms and noting the response of the system through debugging. If the system lacked the tools necessary to perform the given transform, I would add a new rule to enable the program to handle that case. However, this method of testing has the drawback that one cannot decisively conclude if the system is complete or not. I feel that a more mathematical approach by encoding axioms

and proving a suitable degree of completeness is preferable to the approach that I took.

7.2 Control Structure Implementation

One of the most challenging aspects of this project was the integration of all the elements of the control structure. The control structure turned out to be around 2000 lines of code. Furthermore, modularizing the various elements of the propagation engine proved to be quite difficult; and, thus, some code fragments may seem "hacked" in order to get the system to behave properly.

I also learned that pre-packaged rule-based systems like Joshua perform a lot of work for the programmer. In addition to the control elements, Joshua also streamlines the task of encoding rules. While much of the coding for the rules in the ILT system is necessary in order to maintain the generality of the rules, much of the contents of the rules are devoted towards overhead to interface with the forward-propagation mechanism. Thus, much effort was put into reinventing an existing system for small changes.

I chose not to use Joshua as the implementation language in order to build a propagation engine that could understand the weight-driven aspects of the ILT system. Thus, C++ was chosen due to its speed and my experience in the language. However, C++ is not oriented towards rule-based systems, and all the support for propagation had to be coded in a quite general implementation. So, a lot of time spent on the control elements could have been used on improving the rule-systems. Thus, in retrospect, I feel it may have saved me some work and time by experimenting more with Joshua.

8 Conclusion

In conclusion, the ILT system is a robust generalized expert system that can find the inverse Laplace transform for various classes of inputs and relate a detailed report of the process to the user. Furthermore, the scalability of the general

architecture allows the system to be expanded as needed to find the transform of various expressions. The generality and scalability of the robust inference engine ensure that the ILT is a worthwhile knowledge-based system in the field of inverse Laplace transforms.

9 Acknowledgements

All the knowledge about arithmetic expressions was derived by me. The knowledge regarding various Laplace transform properties was taken from the three sources listed in the bibliographic section.

I would like to thank Jacob Eisenstein and Professor Davis for providing prompt comments and suggestions on the project. I would also like to thank my brother Pawan Deshpande for providing me with the formatting for this paper.

Appendix A: Complete Rule Listings

A.1 Always Rules

Number Evaluator
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: numerical expression => number
Added PendingOperations: None
Information: Evaluates any operations contains only numbers as operands

Number Addition
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (+ number1 symbol ... number2) => (+ newNumber symbol ...)
Added PendingOperations: None
Information: Adds two numbers located within the same addition statement

Number Multiplication
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* number1 symbol ... number2) => (* newNumber symbol ...)
Added PendingOperations: None
Information: Multiplies two numbers located within the same multiplication statement

0 Multiplication
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* ... 0 ___) => 0
Added PendingOperations: None
Information: Zeroes any product containing a zero as an operand

Multiplication Identity
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* ... 1 ___) => (* ... ___)
Added PendingOperations: None
Information: Removes a number 1 located within a multiplication statement

Addition Identity
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (+ ... 0 ___) => (+ ... ___)
Added PendingOperations: None
Information: Removes a number 0 located within an addition statement

Division Identity
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (/ A 1) ==> A
Added PendingOperations: None
Information: Simplifies a division-by-1 operation

Add 0-Terms Identity
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (+) => 0
Added PendingOperations: None
Information: Uses the addition identity to simplify the empty sum

Add 0-Terms Identity
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (*) => 1
Added PendingOperations: None
Information: Uses the multiplication identity to simplify the empty sum

Multiply 1-Term
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* exp) => exp
Added PendingOperations: None
Information: Simplifies the product of one number

Add 1-Term
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (+ exp) => exp

Added PendingOperations: None
Information: Simplifies the sum of one number

Addition Simplifier
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (+ ... (+ a b ___ c) ...) => (+ ... a b ___ c ...)
Added PendingOperations: None
Information: Simplifies a nested addition statement

Multiplication Simplifier
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* ... (* a b ___ c) ...) => (* ... a b ___ c ...)
Added PendingOperations: None
Information: Simplifies a nested multiplication statement

Exponent Simplifier 1
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* (P a n) ... (P a m)) => (* (P a (+ n m)) ...)
Added PendingOperations: None
Information: Simplifies the product of two power operations

Exponent Simplifier 2
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* (P a n) ... a) => (* (P a (+ n 1)) ...)
Added PendingOperations: None
Information: Simplifies the product of a base and a power operation

Exponent Simplifier 3
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* a ... a) => (* (P a 2) ...)
Added PendingOperations: None
Information: Simplifies the product of a two expressions into a power operation

Power Simplifier
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (P a 1) => a
Added PendingOperations: None
Information: Returns the first power of an expression

Power Simplifier 2
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (P a 0) => 1
Added PendingOperations: None
Information: Returns the power identity

Power Reducer

Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (P (P a b) c) => (P a (* b c))
Added PendingOperations: None
Information: Applies a nested power simplification

Additive Subtractive Associativity
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (+ (- A B) C) ==> (+ A (* -1 B) C)
Added PendingOperations: None
Information: Simplifies a subtraction statement nested in an addition statement

Division Simplifier
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (/ (* exp ...) (* exp ___)) => (/ (* ...) (* ___))
Added PendingOperations: None
Information: Cancels out expressions in both the numerator and denominator

Addition Commutability
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: ex. (+ 3 s) => (+ s 3)
Added PendingOperations: None
Information: Swaps the location of elements in a addition statement to make the expressions look more consistent

Addition Canceller
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (+ (* a X) (* b X)) => (+ (* (+ a b) X))
Added PendingOperations: None
Information: Factors multiplication statements nested in addition statements

Multiplication Division Reducer
Type: Always Rule Cost: 1.0f Target: Sub-Expressions
Description: (* ... (/ b c)) => (/ (* ... b) c)
Added PendingOperations: None
Information: Simplifies a division statement nested in a multiplication statement

A.2 Lookup Rules

Lookup
Type: Lookup Rule Cost: 0.0f Target: Entire Expression
Description: (/ 1 (P s n)) --> (/ (P t (- n 1)) (G n))
Added PendingOperations: Lookup
Information: Converts an s-Space expression into a t-Space expression

A.3 Guess Rules

Subtraction Flipper
Type: Guess Rule Cost: 100.0f Target: Sub-Expressions
Description: (- a b) => (* -1 (- b a))
Added PendingOperations: None
Information: Reverses the direction of a subtraction statement

Addition Breaker
Type: Guess Rule Cost: 10.0f Target: Entire Expression
Description: (+ A B ... C) => SUM(A B ... C)
Added PendingOperations: None
Information: Uses linearity to break up an addition into a sum of individual transforms

Division Breaker
Type: Guess Rule Cost: 20.0f Target: Entire Expression
Description: (/ (+ A B ... C) D) => SUM((/ A D) (/ B D) ... (/ C D))
Added PendingOperations: None
Information: Uses linearity to break up the addition of fractions into a sum of individual transforms

Quadratic Factorization
Type: Guess Rule Cost: 200.0f Target: Sub-Expressions
Description: (as^2 + bs + c) => factored
Added PendingOperations: None
Information: Applies the quadratic formula on a second degree polynomial and simplifies the result

Division Factorer
Type: Guess Rule Cost: 20.0f Target: Entire Expression
Description: (/ a B) => (/ 1 B) with pending operation (* a)
Added PendingOperations: (Times a)
Information: Uses linearity to simplify a constant divided by an expression

Linearity Factorer
Type: Guess Rule Cost: 10.0f Target: Entire Expression
Description: (* a B ...) ==> (* B ...) with pending operation (* a)
Added PendingOperations: (Times a)
Information: Uses linearity to simplify a constant times an expression

Frequency Shifter
Type: Guess Rule Cost: 100.0f Target: Entire Expression
Description: F(s-a) ==> F(s) with pending operation (Shift (+ a))
Added PendingOperations: (Shift (+ a))
Information: Uses Laplace frequency shifting property

Subtraction Addition Conversion
Type: Guess Rule Cost: 350.0f Target: Sub-Expressions
Description: (- a b) => (+ a (* -1 b))
Added PendingOperations: None
Information: Converts a subtraction into an addition and a multiplication

Scaling
Type: Guess Rule Cost: 200.0f Target: Entire Expression
Description: F(as) => F(s) with pending operation (SCALE a)
Added PendingOperations: None
Information: Uses Laplace frequency scaling property

Partial Fractions
Type: Guess Rule Cost: 200.0f Target: Entire Expression
Description: `(/ A (* (P (- s b ) c ) ... ) ) => (+ (/ d (- s b ) ) (/ d (P (- s b ) 2 ) ) ... )`
Added PendingOperations: None
Information: Applies partial fractions method on a rational function with a factored denominator and simplifies the result

Appendix B: Complete Program Output

This appendix contains a full program output for a rather difficult transform. This section highlights the strengths of ILT including the ability to deal with 3rd degree denominators, quadratic factoring, partial fractions with repeated roots, frequency shifting, linearity, lookups, numeric evaluation and many arithmetic properties. The bold typeface indicates the contents of the file sampleexp2.txt.

```
Enter File Name: sampleexp2.txt
sampleexp2.txt: (/ 1 (* (+ s 1 ) (+ (P s 2 ) (* 6 s ) 9 ) ) )

USER INPUT.
S-SPACE
SUM {
      (/ 1+0I (* (+ s 1+0I ) (+ (P s 2+0I ) (* 6+0I s ) 9+0I ) ) )
            with pending operations:
}

T-SPACE
SUM {
}
Press any key to continue . . .
```

```
Rule 'Addition Commutability'
Which performs: ex. (+ 3 s ) => (+ s 3 )
Was applied on transform pair 0
On the expression: (+ (P s 2+0I ) (* 6+0I s ) 9+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 1+0I (* (+ s 1+0I ) (+ (* 6+0I s ) 9+0I (P s 2+0I ) ) ) )
            with pending operations:
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Addition Commutability'
Which performs: ex. (+ 3 s ) => (+ s 3 )
Was applied on transform pair 0
On the expression: (+ (* 6+0I s ) 9+0I (P s 2+0I ) )
To yield the new transform:

S-SPACE
SUM {
      (/ 1+0I (* (+ s 1+0I ) (+ 9+0I (* 6+0I s ) (P s 2+0I ) ) ) )
            with pending operations:
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Quadratic Factorization'
Which performs: (as^2 + bs + c) => factored
Was applied on transform pair 0
On the expression: (+ 9+0I (* 6+0I s ) (P s 2+0I ) )
To yield the new transform:

S-SPACE
SUM {
      (/ 1+0I (* (+ s 1+0I ) (P (+ s 3+0I ) 2+0I ) ) )
            with pending operations:
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'PARTIALFRACTIONS'
Which performs: (/ A (* (P (- s b ) c ) ... ) ) => (+ (/ d (- s b ) ) (/
d (P (- s b ) 2 ) ) ... )
Was applied on transform pair 0
On the expression: (/ 1+0I (* (+ s 1+0I ) (P (+ s 3+0I ) 2+0I ) ) )
To yield the new transform:

S-SPACE
SUM {
```

```
      (+ (/ 0.25+0I (+ s 1+0I ) ) (/ -0.25+0I (+ s 3+0I ) ) (/ -0.5+0I
(P (+ s 3+0I ) 2+0I ) ) )
            with pending operations:
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Addition Breaker'
Which performs: (+ A B ... C) => SUM(A B ... C)
Was applied on transform pair 0
On the expression: (+ (/ 0.25+0I (+ s 1+0I ) ) (/ -0.25+0I (+ s 3+0I ) )
(/ -0.5+0I (P (+ s 3+0I ) 2+0I ) ) )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ -0.5+0I (P (+ s 3+0I ) 2+0I ) )
            with pending operations:
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Frequency Shifting'
Which performs: F(s-a) ==> F(s) with pending operation (+ a )
Was applied on transform pair 2
On the expression: (/ -0.5+0I (P (+ s 3+0I ) 2+0I ) )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ -0.5+0I (P (+ (- s (+ 3+0I ) ) 3+0I ) 2+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Number Evaluator'
Which performs: numerical expression => number
Was applied on transform pair 2
On the expression: (+ 3+0I )
To yield the new transform:

S-SPACE
SUM {
```

```
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ -0.5+0I (P (+ (- s 3+0I ) 3+0I ) 2+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Additive Subtractive Associativity'
Which performs: (+ (- A B ) C ) ==> (+ A (* -1 B ) C )
Was applied on transform pair 2
On the expression: (+ (- s 3+0I ) 3+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ -0.5+0I (P (+ s (* -1+0I 3+0I ) 3+0I ) 2+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Number Evaluator'
Which performs: numerical expression => number
Was applied on transform pair 2
On the expression: (* -1+0I 3+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ -0.5+0I (P (+ s -3+0I 3+0I ) 2+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Number Addition'
Which performs: (+ number1 symbol ... number2) => (+ newNumber symbol
...)
Was applied on transform pair 2
On the expression: (+ s -3+0I 3+0I )
To yield the new transform:
```

```
S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ -0.5+0I (P (+ s 0+0I ) 2+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Addition Identity'
Which performs: (+ ... 0 ___) => (+ ... ___)
Was applied on transform pair 2
On the expression: (+ s 0+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ -0.5+0I (P (+ s ) 2+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Add 1-Term'
Which performs: (+ exp ) => exp
Was applied on transform pair 2
On the expression: (+ s )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ -0.5+0I (P s 2+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Division Factorer'
Which performs: (/ a B) => (/ 1 B) with pending operation (* a )
Was applied on transform pair 2
```

```
On the expression: (/ -0.5+0I (P s 2+0I ) )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      (/ 1+0I (P s 2+0I ) )
            with pending operations: (Shift (+ 3+0I ) ) (Times -0.5+0I
)
}

T-SPACE
SUM {
}
Press any key to continue . . .


Rule 'Lookup'
Which performs: (/ 1 (P s n)) --> (/ (P t (- n 1)) (G n))
Was applied on transform pair 2
On the expression: (/ 1+0I (P s 2+0I ) )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 1+0I ) )
            with pending operations:
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Frequency Shifting'
Which performs: F(s-a) ==> F(s) with pending operation (+ a )
Was applied on transform pair 0
On the expression: (/ 0.25+0I (+ s 1+0I ) )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ (- s (+ 1+0I ) ) 1+0I ) )
            with pending operations: (Shift (+ 1+0I ) )
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .
```

```
Rule 'Number Evaluator'
Which performs: numerical expression => number
Was applied on transform pair 0
On the expression: (+ 1+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ (- s 1+0I ) 1+0I ) )
            with pending operations: (Shift (+ 1+0I ) )
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Additive Subtractive Associativity'
Which performs: (+ (- A B ) C ) ==> (+ A (* -1 B ) C )
Was applied on transform pair 0
On the expression: (+ (- s 1+0I ) 1+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s (* -1+0I 1+0I ) 1+0I ) )
            with pending operations: (Shift (+ 1+0I ) )
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Number Evaluator'
Which performs: numerical expression => number
Was applied on transform pair 0
On the expression: (* -1+0I 1+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s -1+0I 1+0I ) )
            with pending operations: (Shift (+ 1+0I ) )
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
```

```
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Number Addition'
Which performs: (+ number1 symbol ... number2) => (+ newNumber symbol
...)
Was applied on transform pair 0
On the expression: (+ s -1+0I 1+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s 0+0I ) )
            with pending operations: (Shift (+ 1+0I ) )
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Addition Identity'
Which performs: (+ ... 0 ___) => (+ ... ___)
Was applied on transform pair 0
On the expression: (+ s 0+0I )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I (+ s ) )
            with pending operations: (Shift (+ 1+0I ) )
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Add 1-Term'
Which performs: (+ exp ) => exp
Was applied on transform pair 0
On the expression: (+ s )
To yield the new transform:

S-SPACE
SUM {
      (/ 0.25+0I s )
            with pending operations: (Shift (+ 1+0I ) )
```

```
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Division Factorer'
Which performs: (/ a B) => (/ 1 B) with pending operation (* a )
Was applied on transform pair 0
On the expression: (/ 0.25+0I s )
To yield the new transform:

S-SPACE
SUM {
      (/ 1+0I s )
            with pending operations: (Shift (+ 1+0I ) ) (Times 0.25+0I
)
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Lookup'
Which performs: (/ 1 (P s n)) --> (/ (P t (- n 1)) (G n))
Was applied on transform pair 0
On the expression: (/ 1+0I s )
To yield the new transform:

S-SPACE
SUM {
      DONE
      (/ -0.25+0I (+ s 3+0I ) )
            with pending operations:
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Frequency Shifting'
Which performs: F(s-a) ==> F(s) with pending operation (+ a )
Was applied on transform pair 1
```

```
On the expression: (/ -0.25+0I (+ s 3+0I ) )
To yield the new transform:

S-SPACE
SUM {
      DONE
      (/ -0.25+0I (+ (- s (+ 3+0I ) ) 3+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Number Evaluator'
Which performs: numerical expression => number
Was applied on transform pair 1
On the expression: (+ 3+0I )
To yield the new transform:

S-SPACE
SUM {
      DONE
      (/ -0.25+0I (+ (- s 3+0I ) 3+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Additive Subtractive Associativity'
Which performs: (+ (- A B ) C ) ==> (+ A (* -1 B ) C )
Was applied on transform pair 1
On the expression: (+ (- s 3+0I ) 3+0I )
To yield the new transform:

S-SPACE
SUM {
      DONE
      (/ -0.25+0I (+ s (* -1+0I 3+0I ) 3+0I ) )
            with pending operations: (Shift (+ 3+0I ) )
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
```

```
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Number Evaluator'
Which performs: numerical expression => number
Was applied on transform pair 1
On the expression: (* -1+0I 3+0I )
To yield the new transform:

S-SPACE
SUM {
      DONE
      (/ -0.25+0I (+ s -3+0I 3+0I ) )
             with pending operations: (Shift (+ 3+0I ) )
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Number Addition'
Which performs: (+ number1 symbol ... number2) => (+ newNumber symbol
...)
Was applied on transform pair 1
On the expression: (+ s -3+0I 3+0I )
To yield the new transform:

S-SPACE
SUM {
      DONE
      (/ -0.25+0I (+ s 0+0I ) )
             with pending operations: (Shift (+ 3+0I ) )
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Addition Identity'
Which performs: (+ ... 0 ___) => (+ ... ___)
Was applied on transform pair 1
On the expression: (+ s 0+0I )
To yield the new transform:

S-SPACE
SUM {
      DONE
```

```
      (/ -0.25+0I (+ s ) )
            with pending operations: (Shift (+ 3+0I ) )
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Add 1-Term'
Which performs: (+ exp ) => exp
Was applied on transform pair 1
On the expression: (+ s )
To yield the new transform:

S-SPACE
SUM {
      DONE
      (/ -0.25+0I s )
            with pending operations: (Shift (+ 3+0I ) )
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Rule 'Division Factorer'
Which performs: (/ a B) => (/ 1 B) with pending operation (* a )
Was applied on transform pair 1
On the expression: (/ -0.25+0I s )
To yield the new transform:

S-SPACE
SUM {
      DONE
      (/ 1+0I s )
            with pending operations: (Shift (+ 3+0I ) ) (Times -0.25+0I
)
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .
```

```
Rule 'Lookup'
Which performs: (/ 1 (P s n)) --> (/ (P t (- n 1)) (G n))
Was applied on transform pair 1
On the expression: (/ 1+0I s )
To yield the new transform:

S-SPACE
SUM {
      DONE
      DONE
      DONE
}

T-SPACE
SUM {
      (* (P E (* t (* -1+0I (+ 1+0I ) ) ) ) (* 0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.25+0I (/ (P t (- 1+0I
1+0I ) ) (G 1+0I ) ) ) )
      (* (P E (* t (* -1+0I (+ 3+0I ) ) ) ) (* -0.5+0I (/ (P t (- 2+0I
1+0I ) ) (G 2+0I ) ) ) )
}
Press any key to continue . . .


Which simplifies to:
S-SPACE
SUM {
      DONE
      DONE
      DONE
}

T-SPACE
SUM {
      (* (P E (* t -1+0I ) ) 0.25+0I )
      (* (P E (* t -3+0I ) ) -0.25+0I )
      (* (P E (* t -3+0I ) ) -0.5+0I t )
}
```

References

Edwards, C., Penney, D. (1999). *Elementary Differential Equations with Boundary Value Problems (Fourth Edition)*. USA: Prentice Hall.

Oppenheim, A., Willsky, A. (1997). *Signals and Systems (Second Edition)*. USA: Prentice Hall.

Vibration Data Laplace Transform Table, http://www.vibrationdata.com/Laplace.htm.