MAGICA

A Software Tool for Inferring Types in MATLAB®

(Version 1.0)
OCTOBER, 2002

Pramod G. Joisha
Prithviraj Banerjee

Technical Report No. CPDC-TR-2002-10-004

Contents

1 Introduction

MAGICA (MAthematica system for General-purpose Inferring and Compile-time Analyses) is an extensible inference engine that can determine the types (value range, intrinsic type and array shape) of expressions in a MATLAB program. Written as a Mathematica application, it is designed as an add-on module that any MATLAB compiler infrastructure can use to obtain high-quality type inferences.

About This Document This report only describes MAGICA's capabilities; it doesn't describe the methods, techniques or coding used to achieve them. The intent is to show what the system is capable of, and to demonstrate its usage.

1.1 A Type Inference Using MAGICA

Lines *In[1]* and *Out[1]* below demonstrate a simple interaction with MAGICA through a notebook interface.[a] On line *In[1]*, the MAGICA `type` function object is applied on a representation of the MATLAB expression `sqrt(2)`. MAGICA's response, shown on *Out[1]*, is the inferred type of `sqrt(2)`. In this case, "type" is the expression $\{v, i, s\}$ where v, i and s are the value range, intrinsic type and array shape of `sqrt(2)`. Thus *Out[1]* indicates that the value range of `sqrt(2)` is the point 1.41421, its intrinsic type is the real number designator `$real`, and that its array shape is two-dimensional with unit extents along both dimensions—that is, a scalar shape.

```
In[1]:= type[sqrt[2]]
Out[1]= {1.41421, $real, {⟨1, 1⟩, 2}}
```

1.2 Feature Support

The above is an example of a type inference on a single MATLAB expression. MAGICA can infer the types of whole MATLAB programs comprising an arbitrary number of user-defined functions, each having an arbitrary number of statements. User-defined functions can return multiple values, can consist of assignment statements, the `for` and `while` loops, and the `if` conditional statement. (All these MATLAB constructs are explained in [Mat97].) In addition, MAGICA can handle close to 70 built-in functions in MATLAB. These include important Type II operations[b] like `subsref`, `subsasgn` and `colon` that are used in array indexing and colon expressions. For the most part, the full or nearly the full semantics of a built-in function, as specified in [Mat97], is supported. For instance, subscripts in array indexing expressions can themselves be arrays, and arrays can be complex-valued. Not all of MATLAB's features are currently handled; these include structures, cell arrays and recent additions like function handles.

---

[a] The outputs in this report can be exactly reproduced by typing the code shown against each *In[n]:=* prompt into a notebook interface to version 1.0 of MAGICA, running on Mathematica 4.1.

[b] MATLAB's built-in functions can be classified into one of three groups, based on how the shapes of the outputs are dependent on the shapes of the inputs [JSB00]. Type I built-ins produce outputs whose shapes are completely determined by the shapes of the arguments, if any. Type II built-ins produce an output whose shape is *also* dependent on the elemental values of at least one input. All remaining built-ins fall into the Type III group.

2 Representing MATLAB in MAGICA

MAGICA symbolically represents constructs in MATLAB. An example of this is the Mathematica expression `plus[a, b]`, which is MAGICA's representation of the MATLAB expression `a+b`. On line *In[1]* above, the Mathematica expression `sqrt[2]` was used to denote the MATLAB expression `sqrt(2)`. The idea of functionally representing a MATLAB expression can also be used to denote high-level constructs. For instance, the MATLAB assignment statement `l` $\leftarrow$ `log(-1)`, where `l` is a MATLAB program variable, is represented in MAGICA as shown on line *In[2]* below.

```
In[2]:= assignment[$$lhs :→ l, $$rhs :→ log[-1]]
Out[2]= assignment($$lhs :→ l, $$rhs :→ log(-1))
```

The expression's *head* is `assignment` and this is used to uniquely identify MATLAB assignments. The *tags* `$$lhs` and `$$rhs` serve to identify the assignment's left-hand side and right-hand side. We call `l` and `log[-1]` as *tag values*. A tag value can be any expression; this allows for the representation of arbitrary MATLAB assignments, including the multiple-value assignment [Mat97].

2.1 The Tagging Scheme

In general, MATLAB statements are represented in MAGICA as

$$h[x_1 :\to y_1, x_2 :\to y_2, \ldots, x_n :\to y_n]$$

where the head h serves as a construct identifier, and where the delayed rules [Wol99] $x_i :\to y_i$ $(1 \leq i \leq n)$ stand for *tag-value pairs*. MAGICA places no significance on the position of a tag-value pair; this point should be kept in mind when making new definitions to extend the MAGICA system. An example of the tagging scheme is

if[$$condition $:\to c$**, $$then** $:\to s_t$**, $$else** $:\to s_e$**]**

that represents the `if` conditional statement in MATLAB. Here, c is the `if` statement's test, and s_t and s_e are its `then` and `else` statement bodies. A fair amount of documentation regarding data structure layouts has been coded into MAGICA itself as `usage` messages [Wol99]; this provides a convenient, on-line way of pulling up layout information while interacting with MAGICA.

```
In[3]:= ?if
```

if[$$condition :> c_, $$then :> t_, $$else :> e_]
is the functional equivalent of an if statement in MATLAB. Forms such as
$$condition -> c , $$then -> t and $$else -> e can also be used.

2.2 Extensibility

The decision to represent MATLAB's high-level constructs around the tagging scheme was motivated by extensibility. For instance, if there is a subsequent need to extend the `assignment` data structure by the inclusion of a tag that carries, say, dependency information, this can be easily done without affecting any of the existing MAGICA code.

2.3 A Type Inference on an Assignment Statement

Line *In[4]* below shows an application of the `type` object on the earlier assignment statement.[c] When applied on a single assignment in which the left-hand side is t and the right-hand side is e, `type` generates a list consisting of a single type expression *rewriting rule* [Wol99] of the form $t \to \{v, i, s\}$. The expressions v, i and s are the value range, intrinsic type and array shape of e.

```
In[4]:= type[%2]
Out[4]= {l → {3.14159 ⅈ, $nonreal, {⟨1, 1⟩, 2}}}
```

On *Out[4]*, ⅈ stands for the imaginary unit. The `$nonreal` intrinsic type designator indicates that the elemental value 3.14159ⅈ of `log(-1)` logically belongs to the set $\mathbb{C} - \mathbb{R}$, where $\mathbb{C}$ and $\mathbb{R}$ are the sets of complex and real numbers respectively.

The `type` object may additionally produce side effects; in the above case, it also records the computed v, i and s expressions as upvalues [Wol99] of t. This facilitates later retrieval and reuse of previously computed type expressions. This is also the mechanism by which MAGICA propagates type information from one statement to another. As displayed below, the rank,[d] shape tuple, intrinsic type and value range of t are registered against $\rho(t)$, $\sigma(t)$, $\tau(t)$ and $\upsilon(t)$ respectively.

```
In[5]:= ??l
```

```
Global`l
```

```
ρ(l) ^= 2

σ(l) ^= ⟨1, 1⟩

τ(l) ^= $nonreal

υ(l) ^= 3.14159 ⅈ
```

3 Fibonacci Numbers

MAGICA uses the expression `Sequence[`s_1`, `s_2`, ..., `s_n`]` to denote a sequence of statements. Each s_i $(1 \le i \le n)$ can be an assignment, an `if` conditional, a `for` loop, a `while` loop, a `break` or a `return`. Anything else is taken to be an expression statement.[e] On *In[6]* below, a sequence of statements that calculates the nth Fibonacci number in `fn` is defined and assigned to the Mathematica symbol `stmts`.

[c] The construction `%`n stands for the value computed on the nth output line in Mathematica [Wol99].
[d] In our terminology, the rank of an array is its dimensionality.
[e] Unlike languages such as C, Java, APL and Mathematica itself, MATLAB doesn't consider an assignment as an expression. Hence, a phrase like `(a ← 1)+1` is syntactically illegal in MATLAB.

```
In[6]:= stmts := Sequence[t1 ← sqrt[5], t2 ← plus[1, t1], t3 ←
        mrdivide[t2, 2], t4 ← mpower[t3, n], t5 ← mrdivide[t4, t1],
        fn ← round[t5]]
```

Every statement in the above sequence is an assignment; the construction $l \leftarrow r$ is a shorthand for `assignment[$$lhs :→ l, $$rhs :→ r]` in MAGICA.[f]

As an aside, MAGICA defines a function object called `show` that displays MAGICA data structures in MATLAB syntax. This is useful for visualization.

```
In[7]:= ?show
```

show[s_, opts___] displays a MATLAB structure s as an appropriate MATLAB code fragment, under the control of options in opts.

show[] displays the MATLAB structure assigned to $mfile in the current context.

show has the following options: show$Stream, show$ASCII and show$Notebook. The last two are available only in a notebook – based front – end.

Below we see what the MATLAB progenitor of `stmts` would have looked like.

```
In[8]:= show[stmts]
```

```
%  MATLAB Code Fragment
t1 = sqrt(5);
t2 = 1+t1;
t3 = t2/2;
t4 = t3^n;
t5 = t4/t1;
fn = round(t5);
```

3.1 A Type Inference on Statement Sequences

The MAGICA `type` object can be applied on a sequence of statements to produce a list of rewriting rules; *Out[9]* shows what happens when `type` operates on `stmts`.

```
In[9]:= type[stmts]

Out[9]= {t1 → {2.23607, $real, {⟨1, 1⟩, 2}},
         t2 → {3.23607, $real, {⟨1, 1⟩, 2}},
         t3 → {1.61803, $real, {⟨1, 1⟩, 2}},
         t4 → {(1 + i) ⟦-∞, ∞⟧, $complex, {mpowerST(σ(n), ⟨1, 1⟩), 2}},
         t5 → {(1 + i) ⟦-∞, ∞⟧, $complex, {mpowerST(σ(n), ⟨1, 1⟩), 2}},
         fn → {(1 + i) ⟦-∞, ∞⟧, $complex, {mpowerST(σ(n), ⟨1, 1⟩), 2}}}
```

In the above, we see that the golden ratio computed in `t3` has the value 1.61803, the `$real` intrinsic type, the $\langle 1, 1\rangle$ shape tuple and the rank 2.[g]

[f] The symbol ← can be directly entered into a notebook by typing the key sequence ESC<-ESC.

[g] The inferred rank is redundant in this case; it becomes significant when the shape tuple is symbolic.

3.1.1 Value Ranges in MAGICA

MAGICA's value range inference subsystem is built on Mathematica's interval arithmetic [Wol99]. The value range that MAGICA *conservatively* generates against a MATLAB expression consists of value bounds that *all* elements of that expression honor. (Keep in mind that a MATLAB expression can be a multi-element array.)

Complex Value Ranges MAGICA can denote everything from a value point to a complex value range. The latter are represented using interval arithmetic on real and imaginary subranges. If a MATLAB expression e has the value range

$$[\![r_l, r_h]\!] + [\![i_l, i_h]\!]\,\mathbb{i},$$

it means that the elemental values of e have real and imaginary parts that lie between the inclusive end points r_l and r_h, and i_l and i_h, respectively.[h] Thus *Out[9]* shows that the value ranges of `t4`, `t5` and `fn` are all of the form

$$[\![-\infty, \infty]\!] + [\![-\infty, \infty]\!]\,\mathbb{i}.$$

Note that for these program variables, this is also the best inferable value range. This is because `t4`, `t5` and `fn` are all dependent on `n`, whose value range is unknown.

Value Range Operators MAGICA provides a number of useful "primitives" in connection with value ranges. These operators form the basis for MAGICA's value range inference code. For example, the `nextN` function object returns the IEEE 754 machine *normal* number [Gol91] that comes after a given real number.[i]

```
In[10]:= Names["Type`ValueRange`*"]

Out[10]= {nextN, prevN, υ, υAdd, υApproximation, υDivide, υExp,
          υIntersection, υLimit, υLog, υMemberQ, υMultiply,
          υPointQ, υPower, υRecombine, υUnion, υ$interprocedural}
```

In their ability to handle and produce complex value ranges, value range operators (the third to the second-last in the list on *Out[10]*) go beyond Mathematica's interval arithmetic. This is illustrated below, where `υExp` exponentiates a complex value range.

```
In[11]:= υExp[⟦1, π⟧+i⟦3.1, 4⟧]

Out[11]= ⟦-23.1407, -1.77679⟧ + i ⟦-17.5129, 0.962205⟧
```

If Mathematica's `Exp` built-in object is used instead, the result remains unevaluated because interval arithmetic in Mathematica is only set up for real value ranges.

```
In[12]:= Exp[⟦1, π⟧+i⟦3.1, 4⟧] // N

Out[12]= 2.71828^(⟦1.,3.14159⟧+(0.+1. i) ⟦3.1,4.⟧)
```

[h] MAGICA slightly alters Mathematica's syntax; the way the ⟦ and ⟧ symbols are used is an instance of this. In Mathematica sans MAGICA, these symbols are used in a syntactically different way [Wol99].

[i] `nextN` and `prevN` are used in MAGICA for producing correctly adjusted value range end points.

3.1.2 Intrinsic Types in MAGICA

An intrinsic type in MAGICA denotes a logical set to which *all* elemental values of a MATLAB expression belong. MAGICA's intrinsic types are organized as a lattice $\mathbb{T}$ in which the partial order is value subsumption.[j] This lattice was described in [JB01b] and is reproduced below. MAGICA represents the least and greatest elements of $\mathbb{T}$

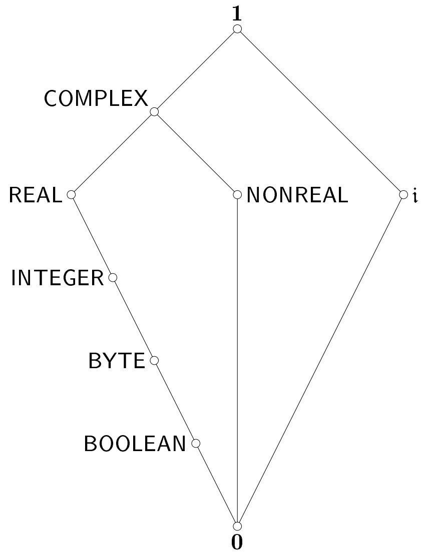

Fig 1. The Lattice $\mathbb{T}$ of Intrinsic Types in MAGICA

by the symbols `$0` and `$1`. The symbols `$boolean`, `$byte`, `$integer`, `$real`, `$complex`, `$nonreal` and `$illegal` similarly denote BOOLEAN, BYTE, INTEGER, REAL, COMPLEX, NONREAL and i respectively.

```
In[13]:= 𝕋
Out[13]= {$0, $boolean, $byte, $integer,
          $real, $nonreal, $complex, $illegal, $1}
```

In [JB01b], a BOOLEAN stood for a 0 or 1, and a BYTE for an 8-bit unsigned integer. The same interpretations have been carried over to MAGICA. As in [JB01b], INTEGER, REAL, COMPLEX and NONREAL signify $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{C} - \mathbb{R}$ in MAGICA (the sets of integers, reals, complexes and strict complexes respectively). [JB01b] did not assume a specific bit width for INTEGER; MAGICA however promotes the intrinsic type of integral values outside the range $[\![-2^{32}+1, 2^{32}-1]\!]$ to REAL, which represents a double-precision number in MAGICA. Lastly, the abstract "illegal" intrinsic type i signifies intrinsic type error situations. Binary comparisons among the symbols work as expected.[k]

```
In[14]:= {$0 ≤ $boolean, $byte ≤ $nonreal, $nonreal ≤ $byte}
Out[14]= {True, False, False}
```

[j] The relation $s \leq t$ means that all values representable by s are also representable by t.

[k] The two `False` values on *Out[14]* together mean that NONREAL and BYTE are not comparable.

3.1.3 Array Shapes in MAGICA

The "array shape" of a MATLAB expression e is the pair $\{\sigma(e), \rho(e)\}$ where $\sigma(e)$ and $\rho(e)$ are the shape tuple and rank of e. On *Out[4]*, we saw that the shape tuple and rank of `log(-1)` were $\langle 1, 1\rangle$ and 2. *Out[9]* shows that the shape tuple of `t4` is

$$\texttt{mpowerST}(\sigma(\mathrm{n}), \langle 1, 1\rangle).$$

MAGICA infers this by considering the right-hand side of `t4` $\leftarrow$ `t3^n`. Because `t3` is inferred to be a scalar from the preceding assignment in `stmts`, and because the shape tuple of `n` is unknown, MAGICA initially computes the shape tuple of `t4` to be

$$\texttt{mpowerST}(\langle 1, 1\rangle, \sigma(\mathrm{n})).$$

The above expression fully describes the shape tuple of `t4` because the matrix power built-in function in MATLAB (invoked either as `mpower(t3, n)` or as `t3^n`) is a Type I operation [JSB00]. Type I operations are characterized by the fact that the shapes of their outputs are fully describable by the shapes of their inputs. For these operations, it is always possible to construct *shape-tuple operators* that map the shape tuples of the inputs to the shape tuples of the outputs [JB01a]. The expression head `mpowerST` above stands for the shape-tuple operator of the `mpower` built-in.

Shape Semantics of the Matrix Power Built-In Function Why is the shape tuple of `t4` symbolic? This is due to the shape semantics of `mpower`:

1. When both `a` and `b` are scalar, `a^b` is the elementary power operation.

2. Otherwise, when `b` is a nonnegative integer, `a^b` is computed by repeated squaring [Mat97]. This means that `a` must be a square matrix for the operation to be valid. The result then has the same shape as `a`. If `b` is a negative integer, MATLAB inverts `a` before proceeding according to this case.

3. For other scalar values of `b`, MATLAB calculates `a^b` using eigenvalues and eigenvectors [Mat97]. Once again, the operation is valid only if `a` is a square matrix, in which case the result has the same shape as `a`.

4. If `b` is a nonscalar square matrix, `a` has to be scalar for the operation to valid. In this case, the shape of the result is the same as that of `b`.

5. Any other shape for `a` or `b` results in an error.

Therefore, because `t3` is a scalar and `n` has an unknown shape, `t3^n` could either have the same shape as `n`, or have an "illegal" shape depending on whether or not `n` is a square matrix. To capture these possibilities, MAGICA returns a symbolic shape tuple.

Canonicalization After arriving at $\texttt{mpowerST}(\langle 1, 1\rangle, \sigma(\mathrm{n}))$, MAGICA rearranges it to $\texttt{mpowerST}(\sigma(\mathrm{n}), \langle 1, 1\rangle)$. This happens because for any two shape tuples $\sigma(s)$ and $\sigma(t)$, $\texttt{mpowerST}(\sigma(s), \sigma(t))$ and $\texttt{mpowerST}(\sigma(t), \sigma(s))$ are *equivalent*—that is, they represent the same shape. This commutative property is coded against `mpowerST` in

MAGICA and causes expressions involving `mpowerST` to be automatically reduced to a *canonical form* [Wol99]. Canonical forms enable MAGICA to detect shape equivalence, which in turn permits the reuse of shape-tuple expressions.

```
In[15]:= typeReuse[%9]

Out[15]= {t1 → {2.23607, $real, {⟨1, 1⟩, 2}},
          t2 → {3.23607, $real, {⟨1, 1⟩, 2}},
          t3 → {1.61803, $real, {⟨1, 1⟩, 2}},
          t4 → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {mpowerST(σ (n), ⟨1, 1⟩), 2}},
          t5 → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {σ (t4), 2}},
          fn → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {σ (t4), 2}}}
```

3.2 Replacing Matrix Power by Array Power

The previous MATLAB Fibonacci code will compute the nth Fibonacci number whenever n is a nonnegative scalar integer. When n is a square matrix, the code will still execute though what is computed in `fn` will probably little resemble a traditional Fibonacci number. And when n is a rectangular matrix or a higher dimensional array, the code will fail due to reasons mentioned earlier.

If we replace the `mpower` built-in by the `power` built-in, we obtain a nice generalization of the code to arbitrary arrays. The `power` built-in, invoked either as `power(a, b)` or as `a.^b`, performs an *elementwise* power operation [Mat97]:

1. When either `a` or `b` is a scalar, the other operand can be any array and the result has the same shape as the other operand.

2. Otherwise, the operation is valid only if `a` and `b` have the same shape. The shape of the result then is the common shape of `a` and `b`.

Substituting `mpower` by `power` allows us to compute an elementwise Fibonacci. The shape of `fn` will then be exactly the shape of `n`, as displayed below.

```
In[16]:= type[t1 ← sqrt[5], t2 ← plus[1, t1], t3 ← mrdivide[t2, 2],
         t4 ← power[t3, n], t5 ← mrdivide[t4, t1], fn ← round[t5]]
Out[16]= {t1 → {2.23607, $real, {⟨1, 1⟩, 2}},
          t2 → {3.23607, $real, {⟨1, 1⟩, 2}},
          t3 → {1.61803, $real, {⟨1, 1⟩, 2}},
          t4 → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {σ (n), ρ (n)}},
          t5 → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {σ (n), ρ (n)}},
          fn → {(1 + ⅈ) ⟦-∞, ∞⟧, $complex, {σ (n), ρ (n)}}}
```

3.3 Implicit Intrinsic Type Coercion

How can we gracefully handle situations in which the elemental values of n aren't nonnegative and integral? One way of doing this is by adopting a conversion policy. For instance, given an n, we could calculate the mth Fibonacci number where

$$m = \lfloor |\mathcal{R}(n)| \rfloor,$$

and where $\mathcal{R}$ extracts the real part of n. The preceding *ad hoc* conversion will at least ensure that for a nonnegative integral n, n and m are equal. When `type` is applied on the modified Fibonacci code, a more refined type inference is obtained against `fn`.

```
In[17]:= type[t1 ← sqrt[5], t2 ← plus[1, t1], t3 ← mrdivide[t2, 2],
         n1 ← real[n], m ← floor[n1], t4 ← power[t3, m], t5 ←
         mrdivide[t4, t1], fn ← round[t5]]
Out[17]= {t1 → {2.23607, $real, {⟨1, 1⟩, 2}},
          t2 → {3.23607, $real, {⟨1, 1⟩, 2}},
          t3 → {1.61803, $real, {⟨1, 1⟩, 2}},
          n1 → {⟦-∞, ∞⟧, $real, {σ(n), ρ(n)}},
          m → {⟦-∞, ∞⟧, $real, {σ(n), ρ(n)}},
          t4 → {⟦0, ∞⟧, $real, {σ(n), ρ(n)}},
          t5 → {⟦0, ∞⟧, $real, {σ(n), ρ(n)}},
          fn → {⟦0, ∞⟧, $real, {σ(n), ρ(n)}}}
```

Note that MAGICA determines the intrinsic type of `fn` to be `$real` rather than `$integer` because the upper bound of the elemental values in `fn` exceeds $2^{32} - 1$.

4 Input Prerequisites

MATLAB programs fed to MAGICA are required to satisfy two prerequisites:

- They have to be in the Static Single-Assignment (SSA) form [CFRW91].
- They have to be in the Single Operator (SO) form.

4.1 Static Single-Assignment Form

The SSA form basically means that program variables have at most a single reaching definition in the representation passed to MAGICA. To support the SSA form, MAGICA also handles binary ϕ functions. Loop-header ϕ functions are required to be represented by the `phi`μ object. All other ϕ functions—namely, those that occur at the end of conditionals and loops—are required to be represented by the `phi`ν object. Due to the structured nature of control flow in the MATLAB language (conditionals, structured loops, and no `goto` statements), ϕ functions in the SSA form will always have lexically preceding definitions, except for loop-header ϕ functions for which one definition will be lexically preceding and the other will be lexically succeeding. MAGICA regards the first operand of a `phi`μ expression as the one having the lexically preceding definition.

The introduction of ϕ functions into a Mathematica representation, and their subsequent removal, is the responsibility of the front-end.

4.2 Single Operator Form

The SO form is an intermediate representation akin to three-address code; a MATLAB expression will be said to be in this form if it is either atomic (that is, either a program variable or a literal program constant), or if it is of the form

$$f(a_1, a_2, \ldots, a_n)$$

where each of the a_i $(1 \leq i \leq n)$ are atomic and where f is a function, either user-defined or built-in. The expression's *arity* is denoted by n, and when n is 0, we have an example of a *niladic* function invocation—that is, an invocation of a function without arguments.[1] For brevity, a MATLAB expression in the SO form will be called a SOF MATLAB expression. If e is a SOF MATLAB expression, then the MATLAB assignment `c` $\leftarrow$ e will also be said to be in the SO form. All expressions and assignments in MATLAB can be cast into the SO form through the introduction of temporaries.

Observe that all of the previous MATLAB code fragments on which `type` was applied were both in the SSA and SO forms.

5 The Hilbert Matrix

The Hilbert matrix H_M is an $M \times M$ matrix in which the (i, j)th element $H_{i,j}$ is

$$\frac{1}{i + j - 1}.$$

A loopy style MATLAB code that computes H_{100} is shown below.

```
In[18]:= stmts := Sequence[M ← 100, H ← zeros[M, M], t1 ← colon[1,
         M], for[$$variable :→ i, $$iterations :→ t1, $$body →
         Sequence[H1 ← phiμ[H, H4], for[$$variable :→ j, $$iterations
         :→ t1, $$body → Sequence[H2 ← phiμ[H1, H3], t2 ← plus[i, j],
         t3 ← minus[t2, 1], t4 ← mrdivide[1, t3], H3 ← subsasgn[H2,
         t4, i, j]]], H4 ← phiν[H1, H3]]], H5 ← phiν[H, H4],
         disp[H5]]
In[19]:= show[stmts]
```

```
%  MATLAB Code Fragment
M = 100;
H = zeros(M, M);
t1 = 1:M;
for i = t1,
    H1 = phiμ(H, H4);
    for j = t1,
        H2 = phiμ(H1, H3);
        t2 = i+j;
        t3 = t2-1;
        t4 = 1/t3;
        H3 = subsasgn(H2, t4, i, j);
    end;
    H4 = phiν(H1, H3);
end;
H5 = phiν(H, H4);
disp(H5)
```

There are four points to note regarding the code fragment:

1. The general form of a MATLAB `for` loop is

   `for` i `=` e`, ... end`

   where i is the loop variable and e is the `for` loop expression. The loop is executed as many times as the number of columns in e. In general, if e has the shape $p_1 \times p_2 \times \cdots \times p_k$ where $k \geq 2$, this equals $p_2 \times \cdots \times p_k$. The iteration count

[1] This terminology is borrowed from APL [PP75].

is determined initially and isn't affected by subsequent modifications of either e (if it is a variable) or i within the body of the loop. With every iteration of the loop, i is set to the vectors that form the successive columns of e.

2. `phi`μ and `phi`ν expressions are used to bring the code to the SSA form.

3. Temporaries like `t2` and `t3` are used to bring the code to the SO form.

4. The `subsasgn` object is used to represent the left-hand side array indexing operation in MATLAB. MAGICA regards the expression

$$\texttt{subsasgn}(O, R, i_1, i_2, \ldots, i_n)$$

as denoting an array A that has the same elements as O except for elements located by the n subscripts (i_1, i_2 and so on till i_n), which are set to elements from R. The `subsasgn` object shares similar semantics with the `Update` operation described in [CFRW91] except for two important departures:

   - Each subscript i_k ($1 \leq k \leq n$) can be an arbitrary array. The locations in A that are set to elements in R are obtained by taking the Cartesian product of the elemental values in the subscripts. If p_k is the number of elemental values in subscript i_k, R is required to have the shape $p_1 \times p_2 \times \cdots \times p_n$. (There are corner cases that need to be handled. See [Mat97] for details.)
   - If the maximum elemental value in i_k exceeds the extent of O along the kth dimension, A has that maximum value for its extent along the kth dimension. New locations created in A due to such expansions are set to 0.

Thus, the effect of the assignment statement

$$\texttt{H3} \leftarrow \texttt{subsasgn(H2, t4, i, j)}$$

is to set elements in `H3` to corresponding elements in `H2`, except for the element at row `i` and column `j` in `H3`, which is set to `t4`. Because `i` and `j` range between 1 and `M`, the possibility of an expansion doesn't exist here. Line *`Out[20]`* below shows the inferences arrived at when `type` is applied on `stmts`.

```
In[20]:= type[stmts]

Out[20]= {M → {100., $byte, {⟨1, 1⟩, 2}},
          H → {0, $boolean, {⟨100, 100⟩, 2}},
          t1 → {⟦1, 100⟧, $byte, {⟨1, 100⟩, 2}},
          i → {⟦1, 100⟧, $byte, {⟨1, 1⟩, 2}},
          H1 → {⟦0, 1.⟧, $real, {⟨100, 100⟩, 2}},
          j → {⟦1, 100⟧, $byte, {⟨1, 1⟩, 2}},
          H2 → {⟦0, 1.⟧, $real, {⟨100, 100⟩, 2}},
          t2 → {⟦2, 200⟧, $byte, {⟨1, 1⟩, 2}},
          t3 → {⟦1, 199⟧, $byte, {⟨1, 1⟩, 2}},
          t4 → {⟦0.00502513, 1.⟧, $real, {⟨1, 1⟩, 2}},
          H3 → {⟦0, 1.⟧, $real, {⟨100, 100⟩, 2}},
          H4 → {⟦0, 1.⟧, $real, {⟨100, 100⟩, 2}},
          H5 → {⟦0, 1.⟧, $real, {⟨100, 100⟩, 2}},
          {Indeterminate, $illegal, {⟨-1, 1⟩, 2}}}
```

The last inference on *`Out[20]`* represents the type of `disp`'s outcome; the shown type attributes reflect the fact that `disp` doesn't return anything.

5.1 An Unknown M for H_M

The reason MAGICA manages to explicitly infer all the array shapes on *Out[20]* is because it has initial information about M. If M were unspecified, MAGICA will only be able to arrive at symbolic expressions for most of the shape tuples.

```
In[21]:= Clear[M]

In[22]:= typeReuse[type[H ← zeros[M, M], t1 ← colon[1, M],
         for[$$variable :→ i, $$iterations :→ t1, $$body → Sequence[H1
         ← phiμ[H, H4], for[$$variable :→ j, $$iterations :→ t1,
         $$body → Sequence[H2 ← phiμ[H1, H3], t2 ← plus[i, j], t3 ←
         minus[t2, 1], t4 ← mrdivide[1, t3], H3 ← subsasgn[H2, t4,
         i, j]]], H4 ← phiν[H1, H3]]], H5 ← phiν[H, H4], disp[H5]]]

Out[22]= {H → {0, $boolean, {⟨floor
            (1/2 (abs (real (subsref (M, 1))) + real (subsref (M, 1)))),
          floor (1/2 (abs (real (subsref (M, 1))) +
               real (subsref (M, 1))))⟩, 2}},
         t1 → {⟦1., ∞⟧, $real, {colonST (1, 1, M), 2}},
         i →
          {⟦1., ∞⟧, $real, {forST(σ (t1)), 2}},
         H1 → {⟦0, 1.⟧, $real, {phiμST(σ (H), σ (H4)), 2}},
         j → {⟦1., ∞⟧, $real, {σ (i), 2}},
         H2 → {⟦0, 1.⟧, $real, {phiμST(σ (H1), σ (H3)), 2}},
         t2 → {⟦2., ∞⟧, $real, {⟨1, 1⟩, 2}},
         t3 → {⟦1., ∞⟧, $real, {⟨1, 1⟩, 2}},
         t4 → {⟦0, 1.⟧, $real, {⟨1, 1⟩, 2}},
         H3 → {⟦0, 1.⟧, $real, {subsasgnST(σ (H2), ⟨1, 1⟩, i, j), 2}},
         H4 → {⟦0, 1.⟧, $real, {phiνST(σ (H1), σ (H3)), 2}},
         H5 → {⟦0, 1.⟧, $real, {phiνST(σ (H), σ (H4)), 2}}}
```

Observe that the shape tuple of H on line *Out[22]* is

$$\langle \lfloor \frac{|R(m)| + R(m)}{2} \rfloor, \lfloor \frac{|R(m)| + R(m)}{2} \rfloor \rangle$$

The term m is the first elemental value in M, and is expressed as `subsref(M, 1)` on *Out[22]*. This inferred shape tuple follows from MATLAB's treatment of extent arguments in array creation functions such as `zeros`, `ones` and `eye`: For an arbitrary M, `zeros(M, M)` is a $\lfloor(|R(m)| + R(m))/2\rfloor \times \lfloor(|R(m)| + R(m))/2\rfloor$ matrix of zeros.

Note that MAGICA still infers `t2`, `t3` and `t4` to be scalars. It deduces this from one key piece of information: The shape tuples of the `for` loop expressions for both `i` and `j` are of the form `colonST[1, 1, M]`. This means that within the body of the innermost loop, both `i` and `j` will be scalars.[m]

5.2 Another Way of Computing H_M

Due to the interpretive overhead associated with executing loops in MATLAB, loopy style code usually performs poorly. An efficient loop-free way of computing H_M that

[m]The `colonST[1, 1, M]` shape tuple expression will be equivalent to either the illegal shape tuple, the $\langle 1, 0\rangle$ empty shape tuple or to the $\langle 1, w\rangle$ shape tuple where w is some positive integer. In the former two cases, both the `i` and `j` loops will not be executed. In the third case, the two loops will be executed and `i` and `j` will be set to scalar integers in the body of the loop.

relies on the language's right-hand side array indexing operation is shown below.

```
In[23]:= stmts := Sequence[t1 ← colon[1, M], t2 ← transpose[t1], t3
         ← ones[1, M], i ← subsref[t2, colon[], t3], j ← subsref[t1,
         t3, colon[]], t4 ← plus[i, j], t5 ← minus[t4, 1], H5 ←
         rdivide[1, t5], disp[H5]]
In[24]:= show[stmts]
```

```
% MATLAB Code Fragment
t1 = 1:M;
t2 = t1.';
t3 = ones(1, M);
i = t2(:, t3);
j = t1(t3, :);
t4 = i+j;
t5 = t4-1;
H5 = 1./t5;
disp(H5)
```

Two new things can be see from the above code fragment:

1. The right-hand side array indexing operation is used to arrive at `i` and `j`.
2. The use of the potentially nonscalar array `t3` as a subscript. In fact, `t3` will be nonscalar for all $M > 1$. Additionally, note the use of the "colon" subscript in `t2(:, t3)` and `t1(t3, :)`, which selects an entire array dimension [Mat97].

The types that are inferred from the new code fragment are shown below.

```
In[25]:= typeReuse[type[stmts]]
Out[25]= {t1 → {⟦1., ∞⟧, $real, {colonST (1, 1, M), 2}},
          t2 → {⟦1., ∞⟧, $real, {transposeST(σ (t1)), 2}},
          t3 → {1, $boolean, {⟨1, floor (1/2 (abs (real (subsref (M, 1))) +
                    real (subsref (M, 1))))⟩, 2}},
          i → {⟦1., ∞⟧, $real, {subsrefST(σ (t2), colon (), t3), 2}},
          j → {⟦1., ∞⟧, $real, {subsrefST(σ (t1), t3, colon ()), 2}},
          t4 → {⟦2., ∞⟧, $real, {plusST(σ (i), σ (j)), 2}},
          t5 → {⟦1., ∞⟧, $real, {σ (t4), 2}},
          H5 → {⟦0, 1.⟧, $real, {σ (t4), 2}}}
```

And as before, if `M` were initially assigned, all shape tuples will be explicitly inferred.

```
In[26]:= type[M ← 100, t1 ← colon[1, M], t2 ← transpose[t1], t3 ←
         ones[1, M], i ← subsref[t2, colon[], t3], j ← subsref[t1,
         t3, colon[]], t4 ← plus[i, j], t5 ← minus[t4, 1], H5 ←
         rdivide[1, t5], disp[H5]]
Out[26]= {M → {100., $byte, {⟨1, 1⟩, 2}},
          t1 → {⟦1, 100⟧, $byte, {⟨1, 100⟩, 2}},
          t2 → {⟦1, 100⟧, $byte, {⟨100, 1⟩, 2}},
          t3 → {1, $boolean, {⟨1, 100⟩, 2}},
          i → {⟦1, 100⟧, $byte, {⟨100, 100⟩, 2}},
          j → {⟦1, 100⟧, $byte, {⟨100, 100⟩, 2}},
          t4 → {⟦2, 200⟧, $byte, {⟨100, 100⟩, 2}},
          t5 → {⟦1, 199⟧, $byte, {⟨100, 100⟩, 2}},
          H5 → {⟦0.00502513, 1.⟧, $real, {⟨100, 100⟩, 2}},
          {Indeterminate, $illegal, {⟨-1, 1⟩, 2}}}
```

5.3 Yet Another Way of Computing H_M

In addition to loops, array indexing operations can also be avoided in the computation of H_M. The following code fragment shows how.

```
In[27]:= stmts := Sequence[M ← 100, t1 ← colon[1, M], t2 ←
         transpose[t1], t3 ← ones[1, M], t4 ← ones[M, 1], i ←
         mtimes[t2, t3], j ← mtimes[t4, t1], t5 ← plus[i, j], t6 ←
         minus[t5, 1], H5 ← rdivide[1, t6], disp[H5]]
In[28]:= show[stmts]
```

```
%  MATLAB Code Fragment
M = 100;
t1 = 1:M;
t2 = t1.';
t3 = ones(1, M);
t4 = ones(M, 1);
i = t2*t3;
j = t4*t1;
t5 = i+j;
t6 = t5-1;
H5 = 1./t6;
disp(H5)
```

Type inferences, with and without an initial assignment to `M`, are shown below.

```
In[29]:= type[stmts]
Out[29]= {M → {100., $byte, {⟨1, 1⟩, 2}},
          t1 → {⟦1, 100⟧, $byte, {⟨1, 100⟩, 2}},
          t2 → {⟦1, 100⟧, $byte, {⟨100, 1⟩, 2}},
          t3 → {1, $boolean, {⟨1, 100⟩, 2}},
          t4 → {1, $boolean, {⟨100, 1⟩, 2}},
          i → {⟦1, 100⟧, $byte, {⟨100, 100⟩, 2}},
          j → {⟦1, 100⟧, $byte, {⟨100, 100⟩, 2}},
          t5 → {⟦2, 200⟧, $byte, {⟨100, 100⟩, 2}},
          t6 → {⟦1, 199⟧, $byte, {⟨100, 100⟩, 2}},
          H5 → {⟦0.00502513, 1.⟧, $real, {⟨100, 100⟩, 2}},
          {Indeterminate, $illegal, {⟨-1, 1⟩, 2}}}
In[30]:= Clear[M]
In[31]:= typeReuse[type[t1 ← colon[1, M], t2 ← transpose[t1], t3 ←
         ones[1, M], t4 ← ones[M, 1], i ← mtimes[t2, t3], j ←
         mtimes[t4, t1], t5 ← plus[i, j], t6 ← minus[t5, 1], H5 ←
         rdivide[1, t6], disp[H5]]]
Out[31]= {t1 → {⟦1., ∞⟧, $real, {colonST (1, 1, M), 2}},
          t2 → {⟦1., ∞⟧, $real, {transposeST(σ (t1)), 2}},
          t3 → {1, $boolean, {⟨1, floor
               (1/2 (abs (real (subsref (M, 1))) + real (subsref (M, 1))))⟩,
             2}}, t4 → {1, $boolean, {⟨floor
               (1/2 (abs (real (subsref (M, 1))) + real (subsref (M, 1)))),
              1⟩, 2}}, i → {⟦0, ∞⟧, $real, {mtimesST(σ (t2), σ (t3)), 2}},
          j → {⟦0, ∞⟧, $real, {mtimesST(σ (t4), σ (t1)), 2}},
          t5 → {⟦0, ∞⟧, $real, {plusST(σ (i), σ (j)), 2}},
          t6 → {⟦-1., ∞⟧, $real, {σ (t5), 2}},
          H5 → {⟦-∞, ∞⟧, $real, {σ (t5), 2}}}
```

6 Bayes Signal Probabilities

In their latest release of MATLAB, announced in July of this year [Matb], The MathWorks incorporated for the first time a capability to analyze MATLAB program types, albeit at *run time*. The code processed in this section is directly from a brochure from The MathWorks that advertises this "JIT-Accelerator technology" [Mata].

```
In[32]:= inputArgs[bayes] ^= {Seq, Matrix, priorProbability};

In[33]:= outputArgs[bayes] ^= {score3};

In[34]:= statements[bayes] ^:= Sequence[Seq ← $init$arg[1], Matrix ←
         $init$arg[2], priorProbability ← $init$arg[3], Pb ←
         mrdivide[1, 4], s1 ← length[Seq], score ← zeros[1, s1], lm
         ← length[Matrix], ls ← minus[s1, lm], s2 ← colon[1, ls],
         for[$$variable → m, $$iterations → s2, $$body →
         Sequence[score1 ← phiμ[score, score2], Pa ←
         priorProbability, k ← minus[m, 1], s3 ← colon[1, lm],
         for[$$variable → n, $$iterations → s3, $$body →
         Sequence[Pa1 ← phiμ[Pa, Pa3], s4 ← plus[k, n], nt ←
         subsref[Seq, s4], t1 ← gt[nt, 0], t2 ← lt[nt, 5], t3 ←
         and[t1, t2], if[$$condition → t3, $$then → Sequence[PbGa ←
         subsref[Matrix, nt, n], s5 ← mtimes[Pa1, PbGa], s6 ←
         minus[1, Pa1], s7 ← mtimes[s6, 0.25], Pb1 ← plus[s5, s7],
         s8 ← mtimes[PbGa, Pa1], Pa2 ← mrdivide[s8, Pb1]]], Pa3 ←
         phiν[Pa1, Pa2]]], Pa4 ← phiν[Pa, Pa3], score2 ←
         subsasgn[score1, Pa4, m]]], score3 ← phiν[score, score2]]

In[35]:= show[statements[bayes]]
```

```
%  MATLAB Code Fragment
Seq = _init_arg(1);
Matrix = _init_arg(2);
priorProbability = _init_arg(3);
Pb = 1/4;
s1 = length(Seq);
score = zeros(1, s1);
lm = length(Matrix);
ls = s1-lm;
s2 = 1:ls;
for m = s2,
    score1 = phiμ(score, score2);
    Pa = priorProbability;
    k = m-1;
    s3 = 1:lm;
    for n = s3,
        Pa1 = phiμ(Pa, Pa3);
        s4 = k+n;
        nt = Seq(s4);
        t1 = nt>0;
        t2 = nt<5;
        t3 = t1&t2;
        if  t3,
            PbGa = Matrix(nt, n);
            s5 = Pa1*PbGa;
            s6 = 1-Pa1;
            s7 = s6*0.25;
            Pb1 = s5+s7;
            s8 = PbGa*Pa1;
            Pa2 = s8/Pb1;
        end;
        Pa3 = phiν(Pa1, Pa2);
    end;
    Pa4 = phiν(Pa, Pa3);
    score2 = subsasgn(score1, Pa4, m);
end;
score3 = phiν(score, score2);
```

6.1 User-Defined MATLAB Functions in MAGICA

User-defined functions in MATLAB are characterized by four essential parts: a name, an input argument list, an output argument list and a function body. These are specified in MAGICA by means of a symbol (transliterated into the Mathematica name space if necessary), and three upvalue expressions. If the symbol f signifies a user-defined function in MATLAB, the expressions `inputArgs[`f`]`, `outputArgs[`f`]` and `statements[`f`]`, recorded as upvalues against f, are used to specify that function's input arguments, output arguments and function body respectively. Lines *In[32]* to *In[34]* above show how this is done for the `bayes` function given in [Mata]. The only differences between the code shown above and that given in [Mata] are:

1. It has additional assignments due to the SSA and SO transformations.

2. It includes a set of dummy assignments against each of the formal parameters at the beginning. The introduction and use of such assignments are a consequence of MAGICA's design: MAGICA uses them to associate the actual parameters at a call site of a user-defined function with the formal parameters of that function. The need for establishing such associations arises when MAGICA propagates type information across user-defined function interfaces.

6.2 A Type Inference on a User-Defined Function

Timings were reported in [Mata] on a call of `bayes` with a 4×20 matrix of doubles for `Matrix`, a 1×912211 matrix of 8-bit integers for `Seq`, and a prior probability of 0.0001 for `priorProbability`. Line *In[36]* below denotes a statement sequence that invokes `bayes` using "randomly" generated inputs that have these shapes. (The manner in which the inputs were created was not mentioned in [Mata]. As we shall see, knowledge of this can impact the inferences that MAGICA makes.)

```
In[36]:= stmts := Sequence[r1 ← rand[1, 912211], r2 ← mtimes[r1, 3],
         r3 ← plus[r2, 1], Seq0 ← fix[r3], Matrix0 ← rand[4, 20],
         priorProbability0 ← 0.0001, score0 ← bayes[Seq0, Matrix0,
         priorProbability0]]
In[37]:= show[stmts]
```

```
%  MATLAB Code Fragment
r1 = rand(1, 912211);
r2 = r1*3;
r3 = r2+1;
Seq0 = fix(r3);
Matrix0 = rand(4, 20);
priorProbability0 = 0.0001;
score0 = bayes(Seq0, Matrix0, priorProbability0);
```

A Not So Random `Seq` When it encounters a user-defined function, MAGICA propagates type information into the function using type information gathered at the call site. In the above, the actual parameters against `Matrix` and `Seq` have been randomly generated, although those generated against `Seq` have been purposefully constructed to lie between 0 and 5. Because of this, MAGICA, which uses symbolic execution to infer types through control structures, figures out that the single conditional in `bayes` will always be executed. This allows all array shapes in `bayes` to be explicitly inferred. This is how `score0`, the output of `bayes`, is also inferred to be a 1×912211 matrix.

```
In[38]:= type[stmts]

Out[38]= {r1 → {⟦0, 1⟧, $real, {⟨1, 912211⟩, 2}},
          r2 → {⟦0, 3.⟧, $real, {⟨1, 912211⟩, 2}},
          r3 → {⟦1., 4.⟧, $real, {⟨1, 912211⟩, 2}},
          Seq0 → {⟦1, 4⟧, $byte, {⟨1, 912211⟩, 2}},
          Matrix0 → {⟦0, 1⟧, $real, {⟨4, 20⟩, 2}},
          priorProbability0 → {0.0001, $real, {⟨1, 1⟩, 2}},
          score0 → {⟦-∞, ∞⟧, $real, {⟨1, 912211⟩, 2}}}
```

The fact that MAGICA manages to infer the shapes of all variables in `bayes` can be verified by calculating the fraction of inferred shapes in it that are explicit.

```
In[39]:= variables[bayes] ^= Cases[{statements[bayes]}, l_ ←
         r_|($$variable → l_) → l, ∞]
Out[39]= {Seq, Matrix, priorProbability, Pb, s1, score, lm, ls,
          s2, m, score1, Pa, k, s3, n, Pa1, s4, nt, t1, t2, t3,
          PbGa, s5, s6, s7, Pb1, s8, Pa2, Pa3, Pa4, score2, score3}
In[40]:= N[Length[Cases[σ /@ #,
         HoldPattern[st[__Integer]]]]/Length[#]]&[variables[bayes]]
Out[40]= 1.
```

Observe that *Out[39]* shows that there are 32 program variables defined in `bayes`—this count includes the three formal parameters and the two loop variables.

```
In[41]:= stmts := Sequence[r1 ← rand[1, 912211], r2 ← mtimes[r1,
         255], Seq0 ← fix[r2], Matrix0 ← rand[4, 20],
         priorProbability0 ← 0.0001, score0 ← bayes[Seq0, Matrix0,
         priorProbability0]]
In[42]:= show[stmts]
```

```
%  MATLAB Code Fragment
r1 = rand(1, 912211);
r2 = r1*255;
Seq0 = fix(r2);
Matrix0 = rand(4, 20);
priorProbability0 = 0.0001;
score0 = bayes(Seq0, Matrix0, priorProbability0);
```

```
In[43]:= type[stmts]

Out[43]= {r1 → {⟦0, 1⟧, $real, {⟨1, 912211⟩, 2}},
          r2 → {⟦0, 255.⟧, $real, {⟨1, 912211⟩, 2}},
          Seq0 → {⟦0, 255⟧, $byte, {⟨1, 912211⟩, 2}},
          Matrix0 → {⟦0, 1⟧, $real, {⟨4, 20⟩, 2}},
          priorProbability0 → {0.0001, $real, {⟨1, 1⟩, 2}},
          score0 → {⟦-∞, ∞⟧, $real, {σ (score3), 2}}}
```

A Random `Seq` If we however construct `Seq` so that its elements span all possible 8-bit values, MAGICA will only explicitly infer some of the shapes in `bayes`. The others will all be symbolic expressions. Still, as shown on *Out[44]* below, MAGICA will explicitly infer close to 60% of the shapes in `bayes`.

```
In[44]:= N[Length[Cases[σ /@ #,
         HoldPattern[st[__Integer]]]]/Length[#]]&[variables[bayes]]
Out[44]= 0.59375
```

7 Adaptive Quadrature by Simpson's Rule

The program processed in this section is a benchmark from the FALCON compiler test suite [FAL]. The benchmark is organized as two input files, one a driver script and the other containing the `adapt` function that does the actual quadrature calculation. The total number of lines across the two input files at the source level is about 79 (excluding comments and empty lines). The source code is shown in the appendix.

Input files that constitute a MATLAB program are called *M-files* in MATLAB parlance. MAGICA currently relies on a custom front-end called MAT℄ to parse M-files to an intermediate form, and to transform that representation to the SSA and SO forms. (MAT℄, not described in this report, is a MATLAB-to-C translator that ultimately compiles MATLAB programs to optimized C versions; it relies on MAGICA to obtain the necessary type information.) Using Mathematica's information hiding context mechanism [Wol99], MAGICA also provides functionality by which *complete* M-file representations, referred to as *M-file contexts*, can be saved and later retrieved. This functionality is used below to load the M-file contexts of the quadrature program. These M-file contexts were automatically created by MAT℄ in an early session of MAGICA.

```
In[45]:= ?load
```

load[x_String] loads the M – file context associated with x . x can also be a string pattern that specifies a set of M – file contexts that need to be loaded.

If x is a previously loaded M – file context, load switches the current M – file context to x . If not, and if x exists on disk, the saved image is loaded. If x is not a previously loaded M – file context and if no image of x exists on disk, a new M – file context corresponding to x is created.

The result of a load is either the name of the last loaded M – file context or Null if one doesn' t exist.

load has the following options : load$Purge and load$Disk.

```
In[46]:= Scan[load[#, load$Disk → True]&, {"drv$adapt`", "adapt`"}]
```

By putting them into separate Mathematica contexts, `load` allows multiple M-file representations, spanning different MATLAB programs, to coexist simultaneously in a single session of MAGICA. Users can switch between M-file contexts by invoking `load`.

7.1 Metrics Reflecting Size

Line *Out[47]* below gives an idea of the size of the `adapt` function: 165 statements with 160 defined variables. The defined variables include the 4 input and 3 output arguments accounted on `Out[48]`. Note that the increase in the overall number of statements and variables is due to the SSA and SO transformations.

```
In[47]:= {Length[Cases[{statements[adapt]},
          _assignment|_if|_for|_while|_break|_disp, ∞]],
          Length[variables[adapt]]}
Out[47]= {165, 160}

In[48]:= {Length[inputArgs[adapt]], Length[outputArgs[adapt]]}
Out[48]= {4, 3}
```

Including the counts shown for the driver script below, the total size of the quadrature program at the stage seen by MAGICA is about 188 statements and 180 variables.[n]

```
In[49]:= load["drv$adapt`"]
Out[49]= adapt`

In[50]:= {Length[Cases[{statements[drv$adapt]},
          _assignment|_if|_for|_while|_break|_disp, ∞]],
          Length[variables[drv$adapt]]}
Out[50]= {23, 20}
```

7.2 A Type Inference on a Program

The Mathematica expression e `//` `Timing` returns $\{t',\ e'\}$ where t' is the time taken by the Mathematica kernel to evaluate e to e'. Line `Out[51]` below shows that the time taken to infer all the types in the quadrature program is 33.66 seconds.

```
In[51]:= type[statements[drv$adapt]] // Timing
Out[51]= {33.66 Second, {_47 t11 → {⟦0, ∞⟧, $real, {⟨1, 6⟩, 2}},
   _57 a1 → {-1, $integer, {⟨1, 1⟩, 2}},
   _59 b1 → {6., $byte, {⟨1, 1⟩, 2}},
   _51 sz_guess1 → {1., $boolean, {⟨1, 1⟩, 2}},
   _55 tol1 → {1.×10^-12, $real, {⟨1, 1⟩, 2}},
   _43 SRmat1 → {⟦-∞, ∞⟧, $real, {σ(_366 SRmat15), 2}},
   _53 quad1 → {⟦-∞, ∞⟧, $real, {σ(_380 quad1), 2}},
   _61 err1 → {⟦0, ∞⟧, $real, {σ(_416 err7), 2}},
   _49 t21 → {⟦0, ∞⟧, $real, {⟨1, 6⟩, 2}}, _650 s → {⟦-∞, ∞⟧,
     $real, {subsrefST(σ(_366 SRmat15), colon()), 2}},
   _651 s → {⟦-∞, ∞⟧, $real,
     {sumST(subsrefST(σ(_366 SRmat15), colon())), 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}}, _652 s →
    {⟦-∞, ∞⟧, $real, {subsrefST(σ(_380 quad1), colon()), 2}},
   _653 s → {⟦-∞, ∞⟧, $real,
     {sumST(subsrefST(σ(_380 quad1), colon())), 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}}, _654 s →
    {⟦0, ∞⟧, $real, {subsrefST(σ(_416 err7), colon()), 2}},
   _655 s → {⟦-∞, ∞⟧, $real,
     {sumST(subsrefST(σ(_416 err7), colon())), 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}},
   _656 s → {⟦-∞, ∞⟧, $real, {⟨1, 6⟩, 2}},
   _657 s → {⟦0, 86400⟧, $integer, {⟨1, 6⟩, 2}},
   _659 s → {⟦0, 86400⟧, $integer, {⟨6, 1⟩, 2}},
   _660 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   _45 _1 t1 → {⟦0, ∞⟧, $real, {⟨1, 1⟩, 2}}}}
```

[n]The driver script statement count considers the invocation of adapt, which returns 3 output values, as three assignments. This is a result of the way MAGICA handles multiple output functions. If these three assignments are treated as one, the statement count reduces to 21.

This measurement, obtained on a 440 MHz UltraSPARC-IIi running Solaris 7 and having 128MB of main memory, is only the time taken for kernel evaluation and does not include the time for the MathLink exchange [Wol99] or other front-end processing.

Line *Out[53]* below shows the fraction of shapes in `adapt` that were explicitly inferred. Among the remaining shapes, which are all symbolic, the fraction that are detected to be equivalent is shown on *Out[55]*.

```
In[52]:= load["adapt`"]

Out[52]= drv$adapt`

In[53]:= N[Length[Cases[σ /@ #,
         HoldPattern[st[__Integer]]]]/Length[#]]&[variables[adapt]]
Out[53]= 0.4

In[54]:= Length[Cases[(# → σ[#])& /@ variables[adapt], HoldPattern[_
         → st[_Symbol]]]]/(Length[variables[adapt]]*(1-%))
Out[54]= 0

In[55]:= Length[Cases[σReuse[(# → σ[#])& /@ variables[adapt]],
         HoldPattern[_ → st[_Symbol]]]]/(Length[variables[adapt]]*(1-
         %%))
Out[55]= 0.59375
```

The σ`Reuse` object on *In[55]* takes a list of rewriting rules of the form $t \to \sigma(t)$, where $\sigma(t)$ is the shape-tuple expression of the variable t, and reuses lexically preceding shape-tuple computations that are symbolically equivalent to lexically succeeding ones. It returns a list of rewriting rules indicating the reuse. Lines *Out[54]* and *Out[55]* above show the difference it can make.

8 The Finite Difference Time Domain Technique

The Finite Difference Time Domain (FDTD) method plays an important role in transient electromagnetic analysis. The example in this section was obtained from a computational electromagnetics course at Chalmers University of Technology [FDT]. The code was chosen because it manipulates three-dimensional arrays and exhibits a lot of array indexing. There are three versions of the FDTD method available at [FDT]—one written using the `diff` built-in function, another using `for` loops, and the third with the `diff` operation expanded out. It is the third version that is processed in this section. The only two differences between the monolithic code given at [FDT] and that used in this report is the inclusion of timing and output commands, and its reorganization into two M-files. The two M-files are shown in § B.

```
In[56]:= Scan[load[#, load$Disk → True]&, {"fdtd`", "drv$fdtd`"}]

In[57]:= {Length[Cases[Join[{statements[drv$fdtd]},
         {statements[fdtd`fdtd]}],
         _assignment|_if|_for|_while|_break|_disp, ∞]],
         Length[Join[variables[drv$fdtd], variables[fdtd`fdtd]]]}
Out[57]= {183, 176}
```

In terms of the total number of statements and variables, line `Out[57]` above shows that the program is about as large as the quadrature program of § 7.[o]

```
In[58]:= type[statements[drv$fdtd]] // Timing

Out[58]= {5.55 Second, {_77 t11 → {⟦0, ∞⟧, $real, {⟨1, 6⟩, 2}},
   _95 Lx1 → {0.05, $real, {⟨1, 1⟩, 2}},
   _97 Ly1 → {0.04, $real, {⟨1, 1⟩, 2}},
   _99 Lz1 → {0.03, $real, {⟨1, 1⟩, 2}},
   _103 Nx1 → {25., $byte, {⟨1, 1⟩, 2}},
   _105 Ny1 → {20., $byte, {⟨1, 1⟩, 2}},
   _107 Nz1 → {15., $byte, {⟨1, 1⟩, 2}},
   _109 nrm1 → {866.025, $real, {⟨1, 1⟩, 2}},
   _101 Nt1 → {128., $byte, {⟨1, 1⟩, 2}},
   _81 Ex1 → {⟦-∞, ∞⟧, $real, {⟨25, 21, 16⟩, 3}},
   _83 Ey1 → {⟦-∞, ∞⟧, $real, {⟨26, 20, 16⟩, 3}},
   _85 Ez1 → {⟦-∞, ∞⟧, $real, {⟨26, 21, 15⟩, 3}},
   _87 Hx1 → {⟦-∞, ∞⟧, $real, {⟨26, 20, 15⟩, 3}},
   _89 Hy1 → {⟦-∞, ∞⟧, $real, {⟨25, 21, 15⟩, 3}},
   _91 Hz1 → {⟦-∞, ∞⟧, $real, {⟨25, 20, 16⟩, 3}},
   _93 Ets1 → {⟦-∞, ∞⟧, $real, {⟨128, 3⟩, 2}},
   _79 t21 → {⟦0, ∞⟧, $real, {⟨1, 6⟩, 2}},
   _649 s → {⟦-∞, ∞⟧, $real, {⟨8400, 1⟩, 2}},
   _650 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}},
   _651 s → {⟦-∞, ∞⟧, $real, {⟨8320, 1⟩, 2}},
   _652 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}},
   _653 s → {⟦-∞, ∞⟧, $real, {⟨8190, 1⟩, 2}},
   _654 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}},
   _655 s → {⟦-∞, ∞⟧, $real, {⟨7800, 1⟩, 2}},
   _656 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}},
   _657 s → {⟦-∞, ∞⟧, $real, {⟨7875, 1⟩, 2}},
   _658 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}},
   _659 s → {⟦-∞, ∞⟧, $real, {⟨8000, 1⟩, 2}},
   _660 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}},
   _661 s → {⟦-∞, ∞⟧, $real, {⟨384, 1⟩, 2}},
   _662 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   {Indeterminate, $illegal, {⟨-1, 1⟩, 2}},
   _663 s → {⟦-∞, ∞⟧, $real, {⟨1, 6⟩, 2}},
   _664 s → {⟦0, 86400⟧, $integer, {⟨1, 6⟩, 2}},
   _666 s → {⟦0, 86400⟧, $integer, {⟨6, 1⟩, 2}},
   _667 s → {⟦-∞, ∞⟧, $real, {⟨1, 1⟩, 2}},
   _75 _1 t1 → {⟦0, ∞⟧, $real, {⟨1, 1⟩, 2}}}}
```

However, the time taken by the kernel to arrive at the type inferences is much lesser because all of the shapes in this program were explicitly inferred. (The processing of symbolic shape-tuple expressions generally contributes to an increase in kernel times.)

```
In[59]:= N[Length[Cases[σ /@ #,
         HoldPattern[st[__Integer]]]]/Length[#]]&[Join[variables[drv$f
         dtd`drv$fdtd], variables[fdtd`fdtd]]]
Out[59]= 1.
```

[o]The `fdtd` function returns seven outputs. If the seven assignments that are generated against the invocation of `fdtd` are counted as one, the total statement count reduces to 181.

9 Availability

MAGICA is currently available for public download from The MAGICA Home Page at

http://www.ece.northwestern.edu/cpdc/pjoisha/MAGICA

9.1 Requirements

To install MAGICA, the following is needed:

- Mathematica version 4.1 or higher,
- A C++ compiler.

A C++ compiler is required because MAGICA relies on external C++ code to realize some of its functionality. (As an example, it uses the C math library function `nextafter` to obtain information on machine normal numbers.)

9.2 Installation

MAGICA has been successfully installed and tested on Solaris 7 and 8, using version 2.95.3 of the `gcc` compiler and version 4.1 of Mathematica. The following are the sequence of installation steps under `tcsh`, the enhanced version of the UNIX C shell.

1. Unzip and untar the downloaded distribution.

   ```
   eagle:~ % gunzip magica.tar.gz
   eagle:~ % tar -xvf magica.tar
   ```

2. Set the `MATHEMATICA` environment variable to the full path of the top directory of the local Mathematica installation. (In Mathematica, this is the string assigned to the `$TopDirectory` system object.) The shell command shown below assumes the existence of Mathematica on the execution path.

   ```
   eagle:~ % setenv MATHEMATICA \
   ? `math -noinit -run 'Print[$TopDirectory]' \\
   ? -run 'Quit[]' | sed -n '$ p'`
   ```

3. Set the `MAGICA` environment variable to the full path of the top directory of the MAGICA installation.

   ```
   eagle:~ % cd MAGICA
   eagle:~/MAGICA % setenv MAGICA `pwd`
   ```

4. Set the `CC` environment variable to the full path of the C++ compiler. If not set, the `install` script (see Step 5 below) will check to see if the `gcc` compiler is present on the execution path and set `CC` to that.

5. Invoke the `install` shell script provided under the `.Mathematica` directory.

```
eagle:~/MAGICA % .Mathematica/install
Using gcc (version 2.95.3) as the C++ compiler ...
        Building syntax-extension ...
        Building wall-clock-time ...
        Building from-file-name ...
        Building normal-numbers ...
```

6. MAGICA is now ready for use. It can be used from a notebook front-end by starting Mathematica with the `-preferencesDirectory` option set. Under MAGICA's preferences directory is an `init.m` file and a POSIX shell script that together "bootstrap" MAGICA. The shell command shown below sets up a shorthand to perform this invocation from anywhere in the directory hierarchy.

```
eagle:~/MAGICA % alias magica "mathematica " \
? "-preferencesDirectory " \
? "$MAGICA/.Mathematica/4.1/ \!*"
```

The `math` script in `.Mathematica/4.1` allows MAGICA to be used from a text front-end. This script can be executed from anywhere in the directory tree.

```
eagle:~/MAGICA % alias magica \
? "$MAGICA/.Mathematica/4.1/math \!*"
```

10 Summary

This report described a software tool called MAGICA that forms a type inference system for the MATLAB programming language. Written in Mathematica, MAGICA infers the value range, intrinsic type and array shape of a MATLAB expression. This report showed the workings of MAGICA by walking through a series of examples of increasing complexity, ranging from single expressions to full programs. Though MAGICA has been shown in an interactive mode, it is possible to use it in batch mode from a custom front-end via the MathLink protocol. Currently, MAGICA is being used this way by MAT2C, a MATLAB-to-C translator that converts MATLAB sources to optimized C.

References

[CFRW91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, and Mark N. Wegman. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph". *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[FAL] *The FALCON Project Home Page* at `http://www.csrd.uiuc.edu/falcon/falcon.html`.

[FDT] *Computational Electromagnetics EEK 170* at `http://www.elmagn.chalmers.se/courses/CEM/`.

[Gol91] David Goldberg. "What Every Computer Scientist Should Know About Floating-Point Arithmetic". *ACM Computing Surveys*, 23(1):5–48, March 1991.

[JB01a] Pramod G. Joisha and Prithviraj Banerjee. "Computing Array Shapes in MATLAB". In *Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*, August 2001.

[JB01b] Pramod G. Joisha and Prithviraj Banerjee. "Correctly Detecting Intrinsic Type Errors in Typeless Languages such as MATLAB". In *Proceedings of the ACM Conference on Array Processing Languages*, pages 6–21, June 2001.

[JSB00] Pramod G. Joisha, U. Nagaraj Shenoy, and Prithviraj Banerjee. "An Approach to Array Shape Determination in MATLAB". Technical Report CPDC–TR–2000–10–010, Center for Parallel and Distributed Computing, Department of Electrical and Computer Engineering, Northwestern University, October 2000.

[Mata] *Accelerating MATLAB: The MATLAB JIT-Accelerator* at `http://www.mathworks.com/mld_accel`.

[Matb] *The MathWorks Announces Release 13 with Major New Versions of MATLAB an d Simulink* at `http://www.mathworks.com/company/pressroom/index.shtml/article/332`.

[Mat97] The MathWorks, Inc. *MATLAB: The Language of Technical Computing*, January 1997. Using MATLAB (Version 5).

[PP75] Raymond P. Polivka and Sandra Pakin. **APL: The Language and Its Usage**. Automatic Computation Series. Prentice-Hall, Inc., 1975. ISBN 0–13–038885–8.

[Wol99] Stephen Wolfram. **The Mathematica Book**. Wolfram Media, Inc., Fourth Edition, 1999. ISBN 0–521–64314–7.

A Adaptive Quadrature by Simpson's Rule

A.1 The **drv_adapt** MATLAB Function

```
function drv_adapt

%%
%% Driver for adaptive quadrature using Simpson's rule.
%%

t1 = clock;

a = -1;
b = 6;
sz_guess = 1;
tol = 1e-12;

[SRmat, quad, err] = adapt(a, b, sz_guess, tol);

t2 = clock;

% Display result.
% disp(SRmat), disp(quad), disp(err);
disp(mean(SRmat(:))), disp(mean(quad(:))), disp(mean(err(:)));

% Display timings.
fprintf(1, 'ADAPT: total = %f\n', (t2-t1)*[0 0 86400 3600 60 1]');
```

A.2 The **adapt** MATLAB Function

```
function [SRmat, quad, err] = adapt(a, b, sz_guess, tol)

SRmat = zeros(sz_guess, 6);
iterating = 0;
done = 1;

h = (b-a)/2; % The step size.
c = (a+b)/2; % The midpoint in the interval.

%% The integrand is f(x) = 13.*(x-x.^2).*exp(-3.*x./2).

Fa = 13.*(a-a.^2).*exp(-3.*a./2);
Fc = 13.*(c-c.^2).*exp(-3.*c./2);
Fb = 13.*(b-b.^2).*exp(-3.*b./2);

S = h*(Fa+4*Fc+Fb)/3; % Simpson's rule.
```

```
SRvec = [a b S S tol tol];

SRmat(1, 1:6) = SRvec;
m = 1;
state = iterating;
while (state == iterating),
      n = m;
      for l = n:-1:1,
          p = l;
          SR0vec = SRmat(p, :);
          err = SR0vec(5);
          tol = SR0vec(6);

          if (tol <= err),
             state = done;
             SR1vec = SR0vec;
             SR2vec = SR0vec;

             a = SR0vec(1); % Left endpoint.
             b = SR0vec(2); % Right endpoint.
             c = (a+b)/2; % Midpoint.

             err = SR0vec(5);
             tol = SR0vec(6);
             tol2 = tol/2;

             a0 = a;
             b0 = c;
             tol0 = tol2;
             h = (b0-a0)/2;
             c0 = (a0+b0)/2;

             %% The integrand is f(x) = 13.*(x-x.^2).*exp(-3.*x./2).

             Fa = 13.*(a0-a0.^2).*exp(-3.*a0./2);
             Fc = 13.*(c0-c0.^2).*exp(-3.*c0./2);
             Fb = 13.*(b0-b0.^2).*exp(-3.*b0./2);

             S = h*(Fa+4*Fc+Fb)/3; % Simpson's rule.

             SR1vec = [a0 b0 S S tol0 tol0];

             a0 = c;
             b0 = b;
             tol0 = tol2;
             h = (b0-a0)/2;
             c0 = (a0+b0)/2;

             %% The integrand is f(x) = 13.*(x-x.^2).*exp(-3.*x./2).
```

```
            Fa = 13.*(a0-a0.^2).*exp(-3.*a0./2);
            Fc = 13.*(c0-c0.^2).*exp(-3.*c0./2);
            Fb = 13.*(b0-b0.^2).*exp(-3.*b0./2);

            S = h*(Fa+4*Fc+Fb)/3; % Simpson's rule.

            SR2vec = [a0 b0 S S tol0 tol0];

            err = abs(SR0vec(3)-SR1vec(3)-SR2vec(3))/10;

            if (err < tol),
               SRmat(p, :) = SR0vec;
               SRmat(p, 4) = SR1vec(3)+SR2vec(3);
               SRmat(p, 5) = err;
            else
               SRmat(p+1:m+1, :) = SRmat(p:m, :);
               m = m+1;
               SRmat(p, :) = SR1vec;
               SRmat(p+1, :) = SR2vec;
               state = iterating;
            end;
         end;
      end;
end;

quad = sum(SRmat(:, 4));

err = sum(abs(SRmat(:, 5)));

SRmat = SRmat(1:m, 1:6);
```

B The Finite Difference Time Domain Technique

B.1 The drv_fdtd MATLAB Function

```
function drv_fdtd

%%
%% Driver for 3D FDTD of a hexahedral cavity with conducting walls.
%%

t1 = clock;

% Parameter initialization.
Lx = .05; Ly = .04; Lz = .03; % Cavity dimensions in meters.
```

```
Nx = 25; Ny = 20; Nz = 15; % Number of cells in each direction.

nrm = norm([Nx/Lx Ny/Ly Nz/Lz]);

Nt = 1024; % Number of time steps.

[Ex, Ey, Ez, Hx, Hy, Hz, Ets] = ...
fdtd(Lx, Ly, Lz, Nx, Ny, Nz, nrm, Nt);

t2 = clock;

% Display result.
% disp(Ex), disp(Ey), disp(Ez);
% disp(Hx), disp(Hy), disp(Hz);
% disp(Ets);
disp(mean(Ex(:))), disp(mean(Ey(:))), disp(mean(Ez(:)));
disp(mean(Hx(:))), disp(mean(Hy(:))), disp(mean(Hz(:)));
disp(mean(Ets(:)));

% Display timings.
fprintf(1, 'FDTD: total = %f\n', (t2-t1)*[0 0 86400 3600 60 1]');
```

B.2 The **fdtd** MATLAB Function

```
function [Ex, Ey, Ez, Hx, Hy, Hz, Ets] = fdtd(Lx, Ly, Lz, ...
 Nx, Ny, Nz, nrm, Nt)

% Physical constants.
eps0 = 8.8541878e-12; % Permittivity of vacuum.
mu0 = 4e-7*pi; % Permeability of vacuum.
c0 = 299792458; % Speed of light in vacuum.

Cx = Nx/Lx; Cy = Ny/Ly; Cz = Nz/Lz; % Inverse cell dimensions.

Dt = 1/(c0*nrm); % Time step.

% Allocate field arrays.
Ex = zeros(Nx, Ny+1, Nz+1);
Ey = zeros(Nx+1, Ny, Nz+1);
Ez = zeros(Nx+1, Ny+1, Nz);
Hx = zeros(Nx+1, Ny, Nz);
Hy = zeros(Nx, Ny+1, Nz);
Hz = zeros(Nx, Ny, Nz+1);

% Allocate time signals.
Ets = zeros(Nt, 3);
```

```
% Initialize fields (near but not on the boundary).
Ex(1, 2, 2) = 1;
Ey(2, 1, 2) = 2;
Ez(2, 2, 1) = 3;

% Time stepping.
for n = 1:Nt,
    % Update H everywhere.
    Hx = Hx+(Dt/mu0)*((Ey(:, :, 2:Nz+1)-Ey(:, :, 1:Nz))*Cz ...
    -(Ez(:, 2:Ny+1, :)-Ez(:, 1:Ny, :))*Cy);
    Hy = Hy+(Dt/mu0)*((Ez(2:Nx+1, :, :)-Ez(1:Nx, :, :))*Cx ...
    -(Ex(:, :, 2:Nz+1)-Ex(:, :, 1:Nz))*Cz);
    Hz = Hz+(Dt/mu0)*((Ex(:, 2:Ny+1, :)-Ex(:, 1:Ny, :))*Cy ...
    -(Ey(2:Nx+1, :, :)-Ey(1:Nx, :, :))*Cx);

    % Update E everywhere except on boundary.
    Ex(:, 2:Ny, 2:Nz) = Ex(:, 2:Ny, 2:Nz)+(Dt/eps0)* ...
    ((Hz(:, 2:Ny, 2:Nz)-Hz(:, 1:Ny-1, 2:Nz))*Cy ...
    -(Hy(:, 2:Ny, 2:Nz)-Hy(:, 2:Ny, 1:Nz-1))*Cz);
    Ey(2:Nx, :, 2:Nz) = Ey(2:Nx, :, 2:Nz)+(Dt/eps0)* ...
    ((Hx(2:Nx, :, 2:Nz)-Hx(2:Nx, :, 1:Nz-1))*Cz ...
    -(Hz(2:Nx, :, 2:Nz)-Hz(1:Nx-1, :, 2:Nz))*Cx);
    Ez(2:Nx, 2:Ny, :) = Ez(2:Nx, 2:Ny, :)+(Dt/eps0)* ...
    ((Hy(2:Nx, 2:Ny, :)-Hy(1:Nx-1, 2:Ny, :))*Cx ...
    -(Hx(2:Nx, 2:Ny, :)-Hx(2:Nx, 1:Ny-1, :))*Cy);

    % Sample the electric field at chosen points.
    Ets(n, :) = [Ex(4, 4, 4) Ey(4, 4, 4) Ez(4, 4, 4)];
end;
```