SLAC-PUB-8732
May 2001

Accelerator Toolbox for MATLAB[1]

A.Terebilo
Stanford Linear Accelerator Center, Stanford University, Stanford, CA 94309 USA

Abstract

This paper introduces Accelerator Toolbox (AT) - a collection of tools to model particle accelerators and beam transport lines in the MATLAB environment. At SSRL, it has become the modeling code of choice for the ongoing design and future operation of the SPEAR 3 synchrotron light source.

AT was designed to take advantage of power and simplicity of MATLAB – commercially developed environment for technical computing and visualization. Many examples in this paper illustrate the advantages of the AT approach and contrast it with existing accelerator code frameworks.

Talk presented at the workshop "Performance Issues at Synchrotron Light Sources"

Advanced Light Source, Lawrence Berkeley Lab, Berkeley CA USA, October 2-4, 2000

[1] Work supported in part by DOE Contract DE-AC03-76SF00515 and Office of Basic Energy Sciences, Division of Chemical Sciences.

1. INTRODUCTION

Computational accelerator physics is a mature field. There are many 'industry-standard' accelerator codes. Collectively, they cover a wide range of accelerator physics problems and numerical methods. Development of new codes or frameworks may seem unnecessary. However, experience shows, that using codes for problems beyond their intended scope is possible but requires significant programming effort and good understanding of the code itself. For this reason, it is a common practice to use several codes for different types of problems in accelerator physics[2]. Having to adopt new accelerator codes on 'ad hoc' basis is inefficient. Especially since most of us are interested in physics of the problem rather than the code itself, command language or the hierarchy of class libraries. There are other inefficiencies such as differences in input file formats between different codes, platform dependency, and (ironically) having to export results to other applications (Mathematica, MATLAB, etc.) for further analysis or visualization.

The motivation behind Accelerator Toolbox (AT) development is to improve *efficiency* and *flexibility* in interactive accelerator modeling. To achieve this AT takes a different approach to many aspects of accelerator computing.

AT abandons the idea of being a stand-alone special purpose software package. It is a MATLAB toolbox – modeled after commercial toolboxes. Functionally, it becomes a part of the MATLAB toolbox set so that AT functions can reuse tools from other toolboxes, for instance optimization and control.

AT is an open-ended collection of tools – MATLAB functions and scripts arranged in functional groups as opposed to one giant engine. Most of AT is written in MATLAB programming language. This code structure greatly simplifies the process of

[2] For example in design of SPEAR3 storage ring [1] we used MAD for optics matching [2] and switched to LEGO [3] for dynamic aperture studies [4]. We had to use yet another code, BETA [5], to analyze the impact of a narrow pole insertion device [6]

adding and sharing new tools (for physicists, by physicists). To avoid the speed drawback of an interpreter environment, computationally intensive routines are written in C/C++ and compiled into MEX-files (binary code executable from within MATLAB).

AT implements *only* computations specific to accelerator physics. All 'housekeeping' tasks and infrastructure such as parsing lattice and commands, programming language, memory management, interactive graphics and many more are already built into MATLAB.

Finally, many accelerator facilities extensively use or plan to use MATLAB. AT can be integrated with such applications to provide an on-line model and useful data structures for accelerator hardware interface.

2. INTRODUCTORY EXAMPLES

To illustrate statements made in the introduction, we give some examples of accelerator modeling with AT. MATLAB users will appreciate the syntax and logic of AT, consistent with other MATLAB toolboxes.

In these and other examples throughout this paper, text that appears in the MATLAB command window is printed in `terminal font`. The user input is preceded by '`>>`' but not the output.

2.1 Loading Lattice

We load an accelerator lattice into workspace by typing the name of the lattice file at MATLAB prompt.

```
>> spear2
** Loading SPEAR lattice in spear2.m **
** Done **
```

Each lattice file is a MATLAB function or script created by the user. It is a sequence of commands that construct a special variable named THERING[3] describing a lattice in MATLAB 'workspace'. More on THERING and lattice data structures in section 3.1-2

2.2 Viewing and Modifying Lattice Parameters

We now can view or manipulate THERING using standard MATLAB syntax. Just like any other workspace variable we can view its basic properties: size, type, etc.

```
>> whos THERING
Name          Size         Bytes  Class
FAMLIST       1x29         58796  cell array (global)
GLOBVAL       1x1            274  struct array (global)
THERING       1x271       401552  cell array (global)
Grand total is 19946 elements using 401552 bytes
```

This output of the '`whos`' command tells us that array THERING has 271 elements, each corresponding to an accelerator lattice element. To inspect the first element, we use the MATLAB '{}' operator:

```
>> THERING{1}
ans =
       FamName: 'AP'
        Limits: [-0.0500 0.0500 -0.0500 0.0500]
    PassMethod: 'AperturePass'
        Length: 0
```

MATLAB tells us that the first element in THERING is an aperture with physical limits -0.05 to 0.05 m in horizontal and vertical planes

[3] AT can handle single pass accelerator structures as well as rings. Variable name THERING is now used for all types of lattices - lines and rings. This may change in the future.

To modify any physical parameter of an element we only need MATLAB commands and operators. For example, to change the limits of the aperture, use {1} to access an element, and '.' (dot) operator to access a field in that element. The new value or array of values appear to the right of the assignment operator '='.

```
>> THERING{1}.Limits = [ -0.05 0.05 –0.02 0.02];
```

In addition to MATLAB syntax, AT provides a number of useful functions to manipulate THERING. AT function `findcells` in the example below finds all elements in the quadrupole family 'QF'.

```
>> QFINDEX = findcells(THERING,'FamName','QF')
QFINDEX =
  Columns 1 through 12
    31   45   47   59   61   77   79   91   93  107  166  180
  Columns 13 through 20
    182  194  196  212  214  226  228  242
```

That is, quadrupoles of the 'QF' family are located in THERING at positions 31, 45, …

AT also provides graphical utilities that allow the user to interactively edit element properties in THERING. Figure 2.1 shows a graphical utility: `intlat`. The user can display the layout of the lattice and select elements and parameters to be modified. A new value typed into a corresponding text box is synchronized with THERING variable in the workspace. The `intlat` tool is written in MATLAB language with the use of MATLAB GUI tools. It is a good starting point for a user who develops his/her own AT GUI applications.

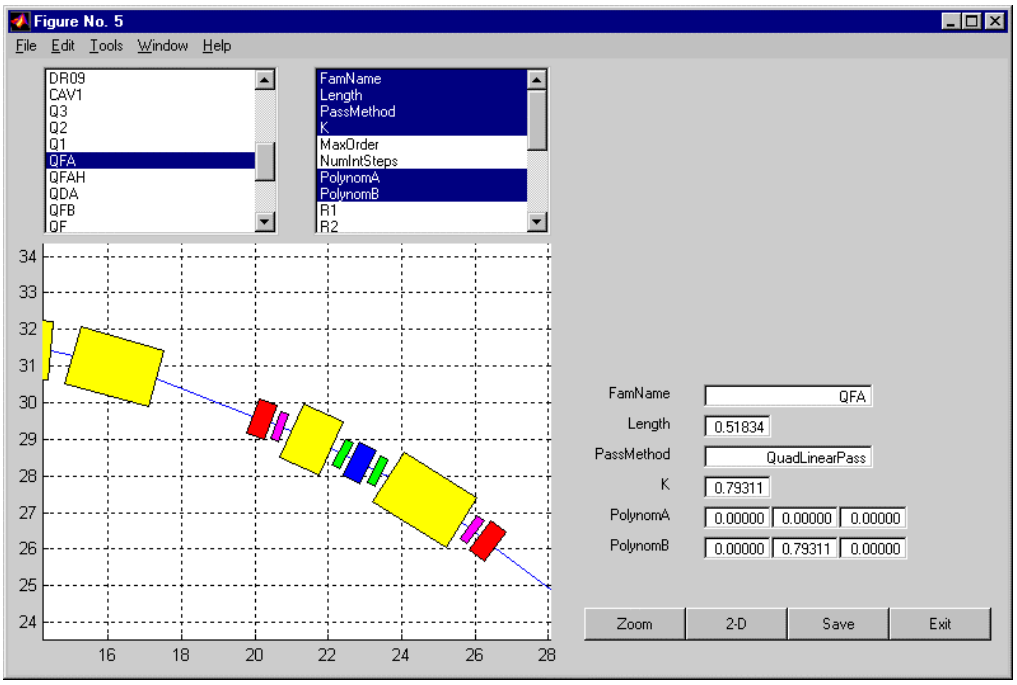

Figure 2.1 Accelerator Toolbox `intlat` – a graphical utility for interactive editing of lattice element properties

2.3 Accelerator Physics Computations

Now we want to compute some accelerator parameters for the lattice we have just loaded. AT function `findm44` finds the linear transverse transfer matrix of a storage ring.

```
>> M = findm44(THERING,0)

M =

    0.6845   30.7722         0         0
   -0.0173    0.6845         0         0
         0         0    0.1253    5.6629
         0         0   -0.1738    0.1253
```

This lattice has no x-y coupling which shows as zero off-diagonal blocks in the transfer matrix. To introduce some coupling we will shift one of the sextupole magnets vertically – in this case element 15 in THERING.

```
>>  THERING{15}
ans =
        FamName: 'SF'
         Length: 0.2334
       MaxOrder: 3
    NumIntSteps: 10
             R1: [6x6 double]
             R2: [6x6 double]
             T1: [0 0 0 0 0 0]
             T2: [0 0 0 0 0 0]
       PolynomA: [0 0 0 0]
       PolynomB: [0 0 1.6769 0]
     PassMethod: 'StrMPoleSymplectic4Pass'
```

'Fields' T1 and T2 describe the shift in particle coordinates due to misalignment of a magnet at entrance and exit. Next two lines effectively move the element vertically by 0.01 m:

```
>> THERING{15}.T1 = [0 0 0.01 0 0 0];
>> THERING{15}.T2 = [0 0 -0.01 0 0 0];
```

Now the transfer matrix is no longer block-diagonal

```
>> findm44(THERING,0)
ans =
    0.6836   30.8164   -0.0686   -1.7660
   -0.0172    0.6861    0.0002    0.0066
   -0.0026    0.1297    0.1251    5.6837
   -0.0046    0.2518   -0.1732    0.1243
```

Another useful AT function `plotbeta` plots beta-functions in a MATLAB figure window and prints tunes in the command window (Figure 2.2):

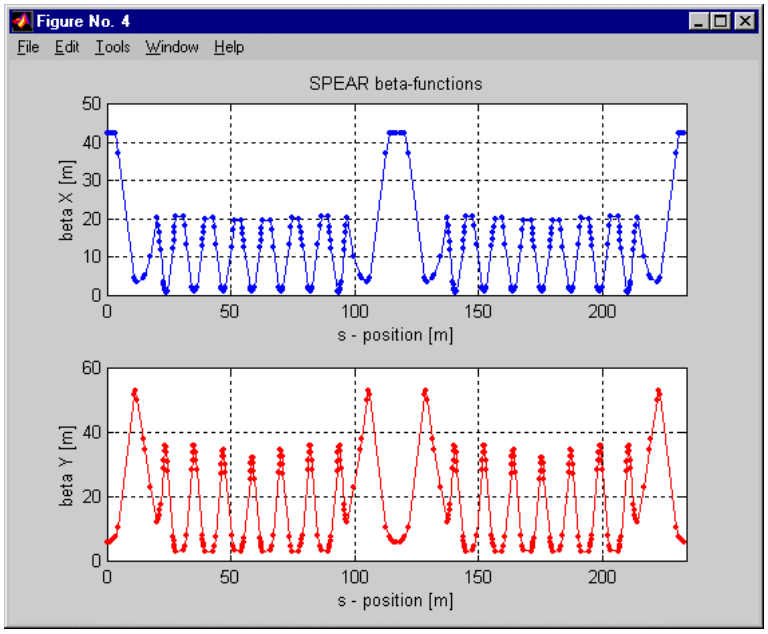

Figure 2.2 AT output of a `plotbeta` command

```
>> plotbeta

tunes =

      0.1290    0.2308
```

2.4 Particle Tracking

AT uses 6-dimensional phase space coordinates

$$\vec{r} = \begin{pmatrix} x \\ p_x \\ y \\ p_y \\ (p - p_0)/p_0 \\ c\tau \end{pmatrix}$$

AT function `ringpass` tracks particles for a specified number of turns: in the example below we track a particle with initial condition $x^{(0)} = y^{(0)} = 0.01m$ for 1000 turns.

```
>> T = ringpass(THERING,[0.01 0 0.01 0 0 0]',1000);
```

The prime symbol in MATLAB denotes a transpose of a vector so that the second argument to `ringpass` function is 6-componet column vector. The result is a 1000 column matrix where each column is a 6-dimentional phase space vector after each of 1000 consequent turns.

```
>> whos T
  Name      Size          Bytes  Class
  T         6x1000        48000  double array
Grand total is 6000 elements using 48000 bytes
```

One of the advatages of AT is that simulation results are immediately available in MATLAB workspace for further analysis or plotting. For example let's use MATLAB plotting functions to display

a. Horizontal position (first component of each column of T) as a function of turn number
b. FFT spectrum of the horizontal motion (Remember, we added some transverse coupling to this lattice - thus multiple lines appear in the spectrum)

```
>> plot(T(1,:),'.')
>> plot(abs(fft(T(1,:))))
```

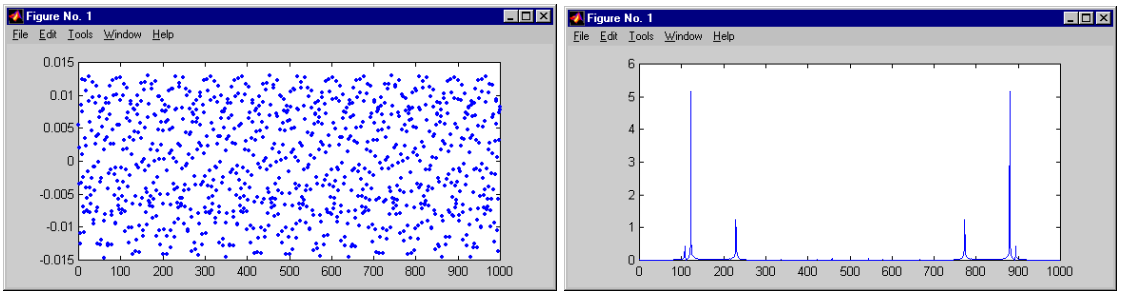

Figure 2.3 Display and analysis of tracking results in MATLAB:

Horizontal (left) position and FFT (right)

2.5 Multiple Particle Tracking and Phase Space Plot

In the style of MATLAB, AT functions can accept vectorized input arguments whenever it adds flexibility. For example `ringpass` can track multiple particles in one call. As an example we will plot horizontal phase space near the 1/3-integer resonance.

First: reload the lattice and fit the strength of quadrupole elements 'QF' and 'QD' to the desired linear tune point near resonance. To start with the original lattice, we use MATLAB 'clear all' command and reload spear2.m

```
>> clear all
>> spear2
** Loading SPEAR lattice in spear2.m **
** Done **
>> fittune2([0.345 0.26], 'QF', 'QD');
```

Using MATLAB syntax we can assemble a multi-column matrix so that each column is a different initial condition with increasing horizontal amplitude:

```
>> X0 = [0.001 0 0 0 0 0]'*(1:15);
```

Now we can pass this matrix of initial conditions as an argument to ringpass and use the MATLAB function `plot` to display horizontal phase space (Figure 2.4).

```
>> XT = ringpassw(THERING,X0,500);
```

`XT` array is the result of tracking of 15 particles for 500 turns.

```
whos XT
  Name      Size          Bytes  Class
  XT        6x7500       360000  double array
Grand total is 45000 elements using 360000 bytes
```

The first 15 columns - `XT(:,1:15)` are the positions of 15 particles after the first turn. The next 15 - `XT(:,16:30)`) are after the second turn etc.

```
>> plot(XT(1,:),XT(2,:),'.');
```

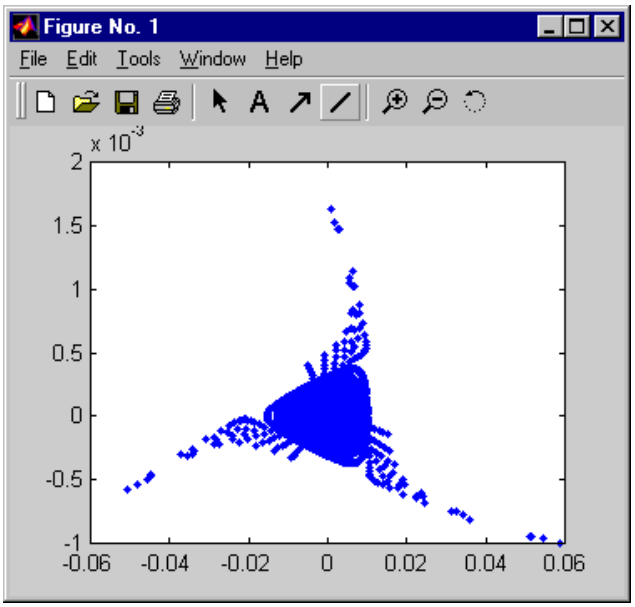

Figure 2.4 Display of the results of tracking of 15 initial conditions in one call to a tracking function `ringpass`

2.6 Programming with AT

One of MATLAB strengths is its high-level programming language that allows the user to make functions and scripts, which then can be immediately called from within MATLAB – just like other MATLAB built-in functions. The following example shows how AT takes advantage of MATLAB programming capabilities.

Let's write a short script `findmcf` that finds momentum compaction factor defined as

$$\alpha = \frac{\delta L / L}{\delta p / p}$$

δL is the path lengthening of an off-momentum orbit with momentum deviation δp. We will make use of AT functions `findorbit4`, `findspos` and `ringpass`. We edit and save a file `findmcf.m`. Comments in a MATLAB files start with `%` symbol.

File `findmcf.m`

```
% Assign a small dP value (normalized to the design momentum)
dP = 0.00001;
% Find transverse fixed point for this dP
FixedPoint = findorbit4(THERING,dP);
% Make an initial condition X0 that starts
%  on this fixed point and has a momentum deviation dP
X0 = [FixedPoint; dP; 0];
% Track this initial condition for 1 turn
% The sixth component of the phase space vector
%  after 1 turn is the path lengthening
X1 = ringpass(THERING,X0);
% Find the total length of the ideal orbit
RingLength = findspos(THERING,length(THERING)+1);
% Calculate alpha parameter
alpha = X1(6)/(dP*RingLength)
```

Now we can call `findmcf` as an AT command from MATLAB prompt:

```
>> findmcf
alpha =
   0.01526317086747
```

2.7 Help for AT functions

MATLAB has a built-in help facility and AT inherits it. Typing `help` followed by a function name prints comments contained in the file with that name. For example this is how we could learn about the `plotbeta` function:

```
>> help plotbeta

  PLOTBETA(RING) finds and plots beta-functions of RING

   It calls FINDOORBI4 and LINOPT which assume a lattice

   with NO accelerating cavities and NO radiation

   PLOTBETA with no arguments uses THERING as the default lattice

   See also PLOTCOD
```

`help` with directory name prints the first line of comments for each file in that directory:

```
>> help D:\MATLABR11\Toolbox\accelerator\atphysics
 CAVITYOFF Turns Cavities OFF:
 CAVITYON Turns Cavities ON:
 FINDM44 finds 4-by-4 transfer matrixes
 LINOPT performs linear analysis of COUPLED!!! lattices
 MCF(RING) calculates momentum compaction factor of RING
 PLOTBETA(RING) finds and plots beta-functions of RING
 RADIATIONOFF Turns classical radiation  OFF:
 RADIATIONON Turns classical radiation  ON:
```

3. LATTICE TOOLS

In accelerator physics the term *lattice* commonly refers to the geometrical arrangement of accelerator elements: magnets, accelerating cavities etc. along with the definition of physical characteristics of each element. All accelerator codes must have some data structures to internally represent lattices.

The central question of most computational accelerator physics problems is - how do the *lattice* parameters affect the dynamics of individual particles and ultimately, *beam* parameters.

AT decouples the task of *lattice definition* from computation of beam parameters by implementing *lattice tools* and *accelerator physics tools* as two separate groups of MATLAB functions ands scripts.

Lattice tools create and manipulate lattice data structures. We discuss lattice tools in this section. Accelerator physics tools simulate particle dynamics and compute beam parameters for a given lattice. Accelerator physics tools are discussed in the next section.

3.1. Lattice Data Structures

Accelerator Toolbox uses a combination of built-in MATLAB data types (a 'cell array'[4] of 'structures') to represent the accelerator lattice.

In MATLAB 'structure', data is arranged in 'fields' and can be accessed by 'field' name. In the example below, QF is a 'structure' that describes a simple quadrupole magnet.

```
>> QF.Name = 'QF';
>> QF.Length = 1;
>> QF.K = 0.55
QF =
  Name: 'QF'
Length: 1
     K: 0.55
```

To change quadrupole strength we need to modify the value of the 'field' K:

```
QF.K=0.5;
```

A 'cell array' in MATLAB is similar to a numeric array but the individual 'cells' can hold data of different types. If we construct in a similar manner 'structures' D1 and D2 for two drift spaces, and 'structure' QD for another quadrupole magnet with negative

[4] Coincidentally, many common terms in accelerator physics such as *cell* or *structure*, also have precise meaning in MATLAB language. To avoid possible confusion, we use single quotes to indicate when a term appears in MATLAB context. For instance '*cell*' means an element of MATLAB 'cell' array, while FODO *cell* refers to a period of an accelerator lattice. Some other terms that may appear in quotes are 'global' 'structure', 'field', …

value of ‘field’ K, we can immediately build a ‘cell array’ to represent one period of a FODO accelerator lattice.

```
>> FODO = {D1 QF D2 QD};
```

Notice how we used MATLAB ‘{}’ syntax for building and indexing cell arrays. We can further combine several FODO periods into a lattice using the MATLAB array concatenation operator ‘[ ]’.

```
>> FODOLATTICE = [FODO FODO FODO FODO];
```

This simple example illustrates an important point: MATLAB already has data types suitable for lattice description - we do not need any special language or any additional code development besides standard MATLAB syntax.

3.2 ‘Global’ Lattice Variables

‘Global’ variables in MATLAB are variables that are shared between functions that use or modify them. It is convenient to use ‘global’ variables for some of the lattice data. To see some of the ‘global’ lattice variables, let’s again load an example lattice and inspect the workspace with `whos` command

```
>> spear2
** Loading SPEAR lattice in spear2.m **
** Done **
>> whos
  Name          Size         Bytes  Class
  FAMLIST       1x29         58796  cell array (global)
  GLOBVAL       1x1            274  struct array (global)
  THERING       1x271       401552  cell array (global)
Grand total is 22551 elements using 460622 bytes
```

3.2.1 THERING

At any given time there can be any number of lattices in the workspace. AT uses one special name THERING for the lattice it is currently working with. THERING is created during the execution of the lattice file.

Many AT functions use this 'global' variable THERING as default argument – for instance `plotbeta` and `intlat` in our introductory examples. The use of 'global' variables also ensures that all changes of lattice parameters made with graphical lattice tools or optics fitting tools are synchronized to the same copy of the lattice.

3.2.2 GLOBVAL

In AT, a 'global' variable GLOBVAL is a 'structure' designed to hold physical parameters that have to be available to AT functions such as design energy E_0. GLOBVAL is created an initialized during lattice declaration.

```
>> GLOBVAL
GLOBVAL =
             E0: 3.0000e+009
    LatticeFile: 'spear2.m'
```

3.2.3 Other AT 'global' variables

LOSSFLAG stores information about particle lost during numerical tracking.

FAMLIST is a 'cell array' that holds a list of families, i.e. groups of elements that share some design properties such as length.

3.3 Lattice files

Commonly, *lattice* files in accelerator codes are text files in a format that the *lattice parser/interpreter* of that particular code can understand. The parser becomes an inextricable part of the code. This traditional approach has disadvantages for both user and developer. The user needs a new lattice definition language for each new accelerator code or a translator program. The developer, often an accelerator physicist, becomes responsible for maintaining the parser module that has nothing to do with physics.

AT takes a different approach. A lattice file in AT is a MATLAB function or script. When executed from command prompt, it creates lattice variables in the MATLAB 'workspace' (THERING and GLOBVAL). There is no strict format specification for a lattice file as long as it correctly produces these lattice variables[5].

An example spear2.m lattice file for SPEAR storage ring is discussed in appendix A.1.

3.4 Element types

Many applications require from an accelerator code the ability to handle many different types of elements and field models. To meet this requirement, an accelerator code has to either include all thinkable element types from the beginning or allow users to define new types with *minimum programming*. There are several steps involved in including or adding a new element type to an accelerator code:

1. Implement a new data type in the code to hold data for the new element type
2. Modify the lattice parser to recognize new element type in the lattice file
3. Implement accelerator physics routines with correct numerical integration methods for the new element type

AT uses MATALB 'structures' to store element data. Physical parameters of an element are stored in separate 'fields' of a 'structure' and can be accessed by the 'field' name. This design makes it easy to create new element types. Consider the following example.

3.4.1 Harmonic Cavity Example

AT models RF cavities assuming a single frequency. Let's make a new type for a *harmonic* cavity [7]. Load an example lattice `spear2rad` that contains a cavity.

```
>> clear all

>> spear2rad

** Loading SPEAR lattice in spear2rad.m **
```

[5] It is even possible, although awkward, in a lattice file to build THERING element-by-element, 'field'-by-'field' as we did in section 3.1. AT provides tools that automatically create 'structures' for elements with consistent 'field' names during the execution of a lattice file.

The last element of THERING is RF cavity with one fundamental frequency. It now has a field ‘Voltage’ that stores the amplitude of the fundamental harmonic.

```
>> THERING{end}

ans =

       FamName: 'CAV1'

        Length: 0

       Voltage: 1600000

     Frequency: 3.585329802858105e+008

    HarmNumber: 280

      PhaseLag: 0

    PassMethod: 'ThinCavityPass'
```

We wish to modify this ‘structure’ to model a harmonic cavity instead. We need to specify the total number of harmonics and their amplitudes. We pick meaningful names for two new fields NumOfHarm and HarmVoltage:

```
>> THERING{end}.HarmVoltage = [1600000 0 800000 0 400000];

>> THERING{end}.NumOfHarm = 5;
```

To change the way AT propagates particles through this new type we also change the ‘field’ PassMethod from `ThinCavityPass` to a new function `ThinHarmCavityPass`[6]. This new function we have to write ourselves.

```
>> THERING{end}.PassMethod = ‘ThinHarmCavityPass’;
```

Now view the changes made to the element:

[6] It is the PassMethod ‘field’ of an element that controls the low-level physics used by AT to propagate particles through that element. We will discuss PassMethod and low-level physics functions in section 4.1

```
>> THERING{end}
ans =
        FamName: 'CAV1'
         Length: 0
        Voltage: 1600000
      Frequency: 3.585329802858105e+008
     HarmNumber: 280
       PhaseLag: 0
     PassMethod: 'ThinHarmCavityPass'
    HarmVoltage: [1600000 0 800000 0 400000]
      NumOfHarm: 5
```

It should be clear from the example that AT does not strictly define element types. Instead, each element has a set of ‘fields’ that store parameters. Making new types in AT only requires adding new ‘fields’ to the element ‘structure’ to store additional parameters. This can be done either from command line or in the lattice file. The new physics, specific to this new type of element, is implemented in a separate function. More on *pass methods* in 4.1.1.

There are a few ‘fields’ that must be the common to all existing element types and newly created ones in order to integrate the new types smoothly into AT. For example all elements must have fields ‘Length’ and ‘PassMethod’.

3.5 Lattice manipulation tools

Any lattice parameter can be modified using MATLAB syntax for ‘cell arrays’ and ‘structures’. AT also provides higher-level lattice tools that modify a group of parameters in several elements in one call.

For example to displace a quadrupole magnet longitudinally by some distance d we need to change the ‘Length’ field of two drifts adjacent to this magnet by $+d$ and $-d$ and also (good AT programming style) check that this operation does not move the

quadrupole to a place already occupied by other magnets. AT function `mvelem` does just that. For example we can move the fifth element in THERING by 1 cm downstream:

```
>> findspos(THERING,5)

ans =

   3.20480000000000
```

This is the current longitudinal position of the magnet in meters. Now move it with `mvelem`:

```
>> mvelem(5,0.01)

>> findspos(THERING,5)

ans =

   3.21480000000000
```

Here are some other lattice manipulation tools with examples:

`cavityon` , `cavityoff` turn on and off all RF cavities in THERING if there are any.

```
>> clear all

>> spear2rad

** Loading SPEAR lattice in spear2rad.m **

** Done **

>> cavityoff

Cavities located at index  [272] were turned OFF
```

`findcells` searches cell arrays of structures such as THERING for elements with specified fields or field values. The next command finds all elements whose 'FamName' equal to 'QF'

```
>> qfindex = findcells(THERING,'FamName','QF')
qfindex =
  Columns 1 through 12
    31  45  47  59  61  77  79  91  93 107 166 180
 Columns 13 through 20
    182 194 196 212 214 226 228 242
```

`settilt`, `setshift` misalign a specified element or a group of elements

```
>> xerror = 0.0001*randn(1,length(qfindex));
>> yerror = 0.0001*randn(1,length(qfindex));
>> setshift(qfindex, xerror, yerror);
```

The first two commands in the above example generate random vectors of horizontal and vertical displacements with 100 μm standard deviation and zero mean for all 'QF' quadrupoles. Then, `setshift` writes these misalignment values to THERING.

4. ACCELERATOR PHYSICS TOOLS

These tools simulate particle dynamics and compute beam parameters for a given lattice. They use AT lattice data structures in the workspace created by AT lattice tools.

4.1 Low-level Physics Tools

Structurally, most accelerator codes are built around a physics engine based on symplectic integration or differential algebra (DA). In the case of symplectic integration, the engine computes trajectories of individual particles element by element – this approach is known as particle tracking or ray tracing. In case of DA the engine computes and analyzes numerical maps of elements or sequences of elements.

AT does not have a single physics engine. Instead, the low-level physics is contained in separate MATLAB functions. The purpose is to make the code structurally

simpler so that the end user can add new physics functionality with minimum knowledge about the rest of the code.

4.1.1 Pass Methods

Special low-level functions that play an important role in AT are the element *pass methods*. Each pass method is implemented in a MATLAB function that numerically propagates particles through an element.

$$\begin{pmatrix} x \\ p_x \\ y \\ p_y \\ \delta \\ c\tau \end{pmatrix}_{Exit} = F\left(ElementData, \begin{pmatrix} x \\ p_x \\ y \\ p_y \\ \delta \\ c\tau \end{pmatrix}_{Entrance} \right)$$

The format for calling pass methods from MATLAB is illustrated in the following example. In our `spear2` lattice, the fifth element in THERING is a quadrupole. We use pass method `QuadLinearPass` to numerically propagate a particle with initial condition $x = y = 1cm$.

```
>> QuadLinearPass(THERING{5},[0.01 0 0.01 0 0 0]')
ans =
   0.00929616821488
  -0.00103587716282
   0.01072074183297
   0.00108625252093
                  0
   0.00000050393612
```

The first argument passed to `QuadLinearPass` is a ‘structure’ containing physical parameters of the magnet: length, strength, misalignment data etc.

```
>> THERING{5}

ans =

        FamName: 'Q2'

         Length: 1.34274000000000

              K: 0.07900900000000

       MaxOrder: 3

    NumIntSteps: 10

       PolynomA: [0 0 0 0]

       PolynomB: [0 0.07900900000000 0 0]

             R1: [6x6 double]

             R2: [6x6 double]

             T1: [0 0 0 0 0 0]

             T2: [0 0 0 0 0 0]

     PassMethod: 'QuadLinearPass'
```

The second argument is a six-dimensional column vector of initial conditions. The output is a vector of six phase space coordinates at the element exit.

4.1.2 Vectorized MATLAB Calling Syntax

The second argument of any *pass method* can also be an N-column matrix where each column represents a different initial condition or a different particle. For example, let `X0` be a matrix of 3 different initial conditions:

```
>> X0 = [0.01 0 0 0 0 0; 0 0 0.01 0 0 0; 0 0 0 0 0.01 0]'
X0 =
   0.01000000000000                  0                   0
                  0                  0                   0
                  0   0.01000000000000                   0
                  0                  0                   0
                  0                  0   0.01000000000000
                  0                  0                   0
```

We can pass `X0` as an argument to a pass method the same way we did a single particle. In the answer, each column is the result of tracking of the corresponding column in `X0`.

```
>> QuadLinearPass(THERING{5},X0)
ans =
   0.00929616821488                  0                   0
  -0.00103587716282                  0                   0
                  0   0.01072074183297                   0
                  0   0.00108625252093                   0
                  0                  0   0.01000000000000
   0.00000024479153   0.00000025914459                   0
```

4.1.3 Pass Methods and Element Models

The computational procedure in each *pass method* assumes some physical *model* of an element. AT includes a number of different model-specific pass methods. The user can independently choose a model for each accelerator element by using one of the available pass methods. Here are some examples of using different models in the case of a quadrupole.

`QuadLinearPass` is based on the exact solution of the Hamiltonian with vector potential A_s proportional to $x^2 - y^2$.

```
>> quadlinearpass(THERING{5}, [0.01 0 0.01 0 0 0]')
ans =
   0.00929616821488
  -0.00103587716282
   0.01072074183297
   0.00108625252093
                  0
   0.00000050393612
```

`DriftPass` assumes a model with no magnetic field. This, obviously, is not a great model for a quadrupole with non-zero gradient.

```
>> driftpass(THERING{5}, [0.01 0 0.01 0 0 0]')
ans =
   0.01000000000000
                  0
   0.01000000000000
                  0
                  0
                  0
```

Notice that the coordinates did not change.

`StrMPoleSymplectic4Pass` implements a fourth-order symplectic integrator [8]. This is equivalent [9] to modeling a quadrupole (or any element with purely transverse magnetic field) as a sequence of transverse kicks and drifts. Notice that the answer is very close but different from the one given by the linear model.

```
>> strmpolesymplectic4pass(THERING{5}, [0.01 0 0.01 0 0 0]')
ans =
   0.00929616840160
  -0.00103587695038
   0.01072074163725
   0.00108625228456
                  0
   0.00000050393603
```

`StrMPoleSymplectic4RadPass` assumes the same type magnetic field as in `StrMPoleSymplectic4Pass` but adds classical radiation. Notice a small change in the fifth component of the phase space vector due to the radiative energy loss

```
>> strmpolesymplectic4radpass(THERING{5}, [0.01 0 0.01 0 0 0]')
ans =
   0.00929616840131
  -0.00103587695025
   0.01072074163756
   0.00108625228446
  -0.00000000063811
   0.00000050393603
```

4.1.4 Element Data – Pass Method Compatibility

Examples in section 4.1.3 show flexibility of AT in defining elements and using different physics models. There is a small price to pay for such flexibility - the user becomes responsible for supplying all necessary element model parameters. The ‘field’ names in the element ‘structure’ must be consistent with the *pass method* used for that element. A *pass method* looks for a set of model parameters in ‘fields’ with specific names. For example:

`DriftPass` expects to find a 'field' named 'Length' in the element 'structure'.

`QuadLinearPass` requires, in addition to 'Length', 'fields' named: 'K' – quadrupole strength, 'R1', 'R2', 'T1', 'T2' – element misalignment parameters.

`StrMPoleSymplecti4Pass` expects to find 'PolynomA', 'PolynomB' which store the coefficients of multipole field expansions and the field 'NumIntSteps' - the number of integration steps for the symplectic integrator. If a *pass method* can not find a field with the exactly matching name, execution stops and MATLAB displays error messages.

4.1.5 Tracking through Sequences of Elements

When tracking through sequences of elements or tracking multiple turns in rings we want to loop over elements and set in advance which the pass method to use for each element. For this purpose each element 'structure' has a 'field' named 'PassMethod'. For example:

```
>> THERING{2}.PassMethod
ans =
DriftPass
>> THERING{3}.PassMethod
ans =
QuadLinearPass
```

One easy way to make a loop over elements is to write a simple function in MATLAB language using `feval` command.

File: `testloop.m`

```
function xout = testloop(xin)
global THERING
xout = xin;
for i = 1:length(THERING)
  METHOD = THERING{i}.PassMethod;
```

```
  xout = feval(METHOD,THERING{i},xout);
end
```

This function tracks one or several initial conditions for one turn through THERING.

We again use a matrix with three different initial conditions `X0`.

```
>> X0 = [ 0.01 0 0 0 0 0; 0 0 0.01 0 0 0; 0 0 0 0 0.01 0]';
```

Now type from MATLAB command line:

```
>> testloop(X0)
ans =
   0.00686446690158  -0.00008290570276   0.01229027878633
  -0.00017504919046   0.00000283372661   0.00068644957948
                   0   0.00134432591088                  0
                   0  -0.00172434135644                  0
                   0                   0   0.01000000000000
   0.00072108772832  -0.00011909976543   0.03383980184588
```

This is a good example to illustrate the use of the 'PassMethod' field. It is not the most efficient: MATLAB loops that are inherently slow since it is an interpreter environment. Accelerator Toolbox has a few tools for fast tracking through sequences. These tools were written in C and compiled into mex-files to achieve native execution speed.

`RingPass` tracks through a ring for a number of turns. We used it in one of the examples in section 2.4.

`LinePass` tracks through a sequence of elements only once but it can return intermediate results at specified points. For example if we were interested in particle's coordinates when it entered elements 1, 100 and 200 we would use:

```
>> linepass(THERING,[0.01 0 0.01 0 0 0]',[1 100 200])

ans =

   0.01000000000000  -0.00181470789175   0.00149615901159
                  0   0.00101651176382  -0.00070670061099
   0.01000000000000   0.02536745801686   0.00890811778247
                  0   0.00449985303116   0.00434806288090
                  0                  0                  0
                  0  -0.00046794511453   0.00013252083694
```

4.2 High-Level Physics tools

Using MATLAB scripting features, AT builds a hierarchy of high-level accelerator physics tools. These functions and scripts compute accelerator and beam parameters. They are built on top of low-level pass methods that hide the details of individual particles dynamics through elements. High-level tools make use of other functions in AT and MATLAB's own toolboxes. Most high-level tools are written in MATLAB programming language.

An example in this section further illustrates the structure of AT as an open-ended collection of interacting MATLAB functions and scripts. A common problem of fitting optics parameters (such as tunes) may involve a number of intermediate steps.

- Define a lattice
- Find closed orbit
- Find a transfer matrix – a linear approximation to the full one turn map of a storage ring near closed orbit
- Compute tunes from (possibly coupled) transfer matrix
- Numerically invert the relationship between free parameters such as strength of two or more quadrupole families and the target parameters – tunes.

Separate AT tools can accomplish these tasks. Figure 4.1 shows the functions involved and their dependencies. We have already seen command line examples of most of these high-level tools in 2.3–2.5.

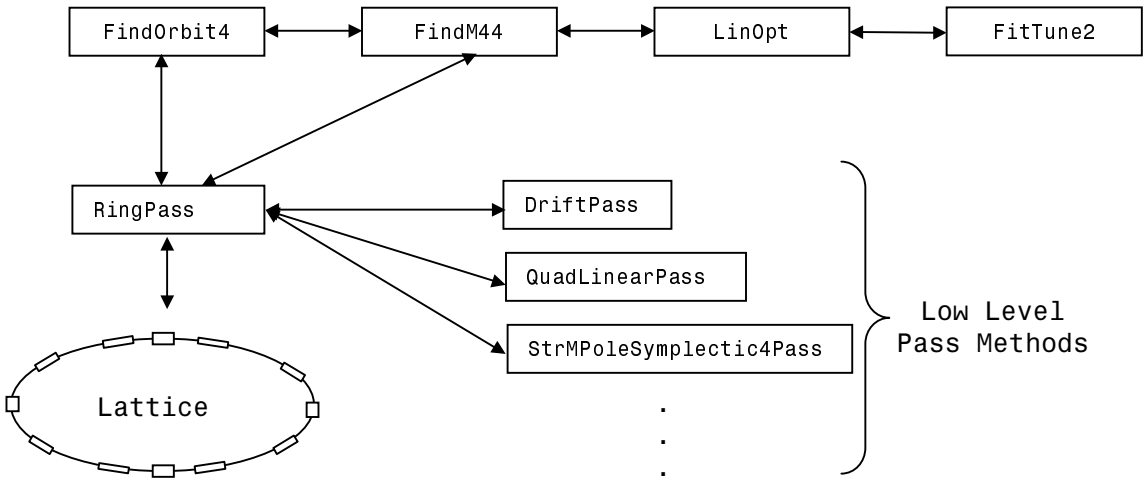

Figure 4.1 AT functions used in linear tunes fitting example

5. DEVELOPMENT AND DISTRIBUTION

5.1 Availability

Accelerator Toolbox is publicly available for download from the AT web page

http://www-ssrl.slac.stanford.edu/at/

It requires MATLAB version 5 or higher.

5.2 Code Development

Developing and maintaining large-scale, non-commercial scientific software is a demanding task. Historically it has been a challenge for physicists and programmers who wrote accelerator codes. One of the key ideas of AT is to outsource most of this work - everything that is not physics.

From the beginning, AT meant to simplify code development and user participation. MATLAB language and open-ended, modular toolbox structure allow users to independently contribute to any of the AT tool groups discussed in this paper:

- Lattice Tools and GUIs
- Pass methods for new element models
- High-level physics tools
- Modeling and control applications

New user-developed tools that fit into existing AT framework can be shared and later included in the code distribution.

ACKNOWLEDGEMENTS

An important piece of work that has preceded AT in systematically using MATLAB for accelerator modeling was TRACY ML [10] by H. Nishimura, LBNL. Symplectic integrator pass methods such as `StrMPoleSymplectic4Pass` use C translations of PASCAL routines used in the accelerator simulation code TRACY [11] by J. Bengtsson, E. Forest and H. Nishsimura. TRACY was also used as a benchmark for AT testing. The author thanks J.Corbett, SLAC, for helpful feedback that results from his work on AT and MATLAB applications for SPEAR3 light source. The author also thanks the accelerator physics group at Advanced Light Source, LBNL: H. Nishimura, G. Portman, D. Robin, C. Steier, for the ongoing collaboration in using AT and contributions made to the code.

REFERENCES

1. R. Hettel, et al “Design of the SPEAR 3 Light Source”, Proc. of European Particle Accelerator Conference, EPAC2000, Vienna, Austria (2000).
2. J. Corbett, et al, "Design of the SPEAR 3 Magnet Lattice", Proc. of European Particle Accelerator Conference, EPAC98, Stockholm, Sweden, June 22-26, 1998. See also SLAC-PUB-7882 (1998).
3. Y. Cai, et al “LEGO: a Modular Accelerator Design Code”, IEEE Proc. of Particle Accelerator Conference PAC97, Vol.2, p.2583 (1998)

4. J. Corbett, Y. Nosochkov, J. Safranek, "Dynamic Aperture Studies for SPEAR 3", Proc. of Particle Accelerator Conference PAC99, (1999). SLAC-PUB-8232. See also SLAC-PUB-7965 (1999).
5. L. Farvacque, JL. Laclare, A. Ropert, “BETA User’s Guide”, ESRF-SR/LAT-88-08, (1989).
6. J. Safranek, et al “Nonlinear Dynamics in SPEAR Wigglers”, Proc. Of European Particle Accelerator Conference, EPAC2000, Vienna, Austria (2000).
7. A. Hofmann and S. Myers, “Beam Dynamics in a double RF system”, In Proc. Of the 11-th International Conference on High Energy Accelerators, pp 610-614, (1980)
8. R.D. Ruth, “A Canonical Integration Technique,” IEEE Trans. Nuclear Science, NS-30, p2669 (1983).
9. See discussion (section 3.3.1) in E. Forest, “Beam Dynamics, New Attitude and Framework”, Harwood Academic Publishers (1998)
10. H. Nishimura,"Introduction to TracyML", LBNL ALS/LSAP-246 (1998). See also H. Nishimura, “Matlab-like environment for accelerator modeling and simulation”, Presented at International Computational Accelerator Physics Conference ICAP98 Monterey, CA, USA (1998).
11. H. Nishimura, ”TRACY, a Tool for Accelerator Design and Analysis”, Proc. Of European Particle Accelerator Conference EPAC 88, p803 (1989)

APPENDIX A.1

This appendix explains the lattice definition convention in Accelerator Toolbox.

A lattice file in AT is a MATLAB function or script written by the user. When executed, it creates lattice data structures in the MATLAB workspace. By convention, the array that describes a lattice is a 'cell array' where each 'cell' represents a lattice element. Each 'cell' is a 'structure' with a set of consistent 'field' names. Some 'fields' such as 'Length' are common to all element types. Some are specific to a physical model chosen for that element. For example:

`Length`	- physical length [m]
`K`	- quadrupole strength for quadrupole and dipole magnets
`BendingAngle`	- bending angle in dipole magnets
`Limits`	- physical aperture limits, a 4-component vector
`PassMethod`	- name of the function to be used for numerical tracking

The advantages of this convention are:

- No need for a special-purpose language to define lattices
- No need for lattice file parser/interpreter program – MATLAB interpreter serves this purpose
- Flexibility to define additional element parameters with minimum programming

Note, that we could create data structures like this using a few built-in MATLAB commands. For example, we could write a lattice file that declares elements one-by-one like this:

```
THERING{1} = struct('FamName','AP','Length',0,...
   'Limits',[-0.05 0.05 -0.02 0.02],'PassMethod','AperturePass');

THERING{2} = struct('FamName','DR01','Length',1.3448,...
   'PassMethod','DriftPass');

THERING{3} = struct('FamName','QF3','Length', . . .);
```

A lattice created in this way would have to contain an entry for each element and may get too long for practical use. The solution is to group elements that are similar by design and function, into families. Families are declared first. The lattice is than constructed from representatives of different families using MATLAB array syntax '{}' and '[]'.

The following example lattice definition file `spear2.m` uses this approach. Its format is similar to lattice file formats of many other accelerator modeling codes. We will now analyze its parts:

File spear2.m (beginning)

```
function spear2
% Simplified lattice definition file for SPEAR-II storage ring
% Created 11/21/99

global FAMLIST THERING GLOBVAL

GLOBVAL.E0 = 3e9;
GLOBVAL.LatticeFile = 'spear2.m';

FAMLIST = cell(0);

disp(' ');
disp('** Loading SPEAR lattice in spear2.m **');
```

The first line with MATLAB keyword 'function' identifies this file as a function with no arguments and no return values – meaning all variables created during its execution are 'local' unless explicitly declared 'global'. Variables `FAMLIST THERING GLOBVAL` are `'global'`. Lines preceded by % are comments. MATLAB will display the top comment lines when 'help' is called. The next block is a description of different element families.

File spear2.m (continued)

```
% Begin Element Families

AP = aperture('AP', [-0.05, 0.05, -0.05, 0.05],'AperturePass');

DR01   =  drift('DR01' ,1.344800,'DriftPass');
DR02   =  drift('DR02' ,0.860000,'DriftPass');
DR03   =  drift('DR03' ,6.413180,'DriftPass');
DR04   =  drift('DR04' ,0.611890,'DriftPass');
DR04A  =  drift('DR04A',0.617123,'DriftPass');
DR05   =  drift('DR05' ,2.823700,'DriftPass');
DR06A  =  drift('DR06A',0.151205,'DriftPass');
DR06B  =  drift('DR06B',0.229935,'DriftPass');
DR07A  =  drift('DR07A',0.229948,'DriftPass');
DR07B  =  drift('DR07B',0.151205,'DriftPass');
DR08A  =  drift('DR08A',0.151205,'DriftPass');
DR08B  =  drift('DR08B',0.227335,'DriftPass');
DR09   =  drift('DR09' ,2.981660,'DriftPass');

Q3   =  quadrupole('Q3'  , 1.00000, 0.000000,'QuadLinearPass');
Q2   =  quadrupole('Q2'  , 1.34274, 0.079009,'QuadLinearPass');
Q1   =  quadrupole('Q1'  , 0.51834,-0.259585,'QuadLinearPass');
QFA  =  quadrupole('QFA' , 0.51834, 0.793115,'QuadLinearPass');
QFAH =  quadrupole('QFAH', 0.25917, 0.793115,'QuadLinearPass');
QDA  =  quadrupole('QDA' , 0.51834,-0.654627,'QuadLinearPass');
QFB  =  quadrupole('QFB' , 0.51834, 0.516968,'QuadLinearPass');
QF   =  quadrupole('QF'  , 0.51834, 0.449896,'QuadLinearPass');
QD   =  quadrupole('QD'  , 0.51834,-0.669244,'QuadLinearPass');

SF  = sextupole('SF' , 0.23335, 1.67687,'StrMPoleSymplectic4Pass');
SDA = sextupole('SDA', 0.23335,-1.29030,'StrMPoleSymplectic4Pass');
SDB = sextupole('SDB', 0.23335,-1.29030,'StrMPoleSymplectic4Pass');

BB  =  rbend('BB'  ,2.35785400,  ...
                  0.1848, 0.0924, 0.0924, 0,'BendLinearPass');
B   = rbend('B'   ,1.17766900,   ...
                       0.0924, 0.0462, 0.0462, 0,'BendLinearPass');
```

In this block, `aperture`, `drift`, `quadrupole`, `sextupole` and `rbend` are functions implemented in AT to create element families and initialize their data fields. Each function updates the list of declared families `FAMLIST` and returns an integer index by which this family is entered in `FAMLIST`.

The next block creates an ordered list of elements ELIST. Elements in this list are integer references to families in FAMLIST declared in the previous block. In this example ELIST is built from pieces (sub-lines) SWSE, NENW and AP using MATLAB concatenation syntax '[]'. It is easy to symbolically handle sub-line repetitions and order inversion using standard syntax or additional AT tools such as reverse.

File spear2.m (continued)

```
% Begin Lattice

SWSE =[ DR01 Q3 DR02 Q2 DR03 Q1 DR04 BB DR04A BB DR05 QFA DR06A ...
      SF DR06B B DR07A SDA DR07B QDA DR08A ...
      SDA DR08B BB DR08B SF DR08A QFB DR09 ...
      QF DR04 BB DR08B SDB DR08A QD DR08A SDB DR08B BB DR08B ...
      SF DR08A QF DR09 QF DR04 BB DR08B SDA DR08A QD ...
      DR08A SDA DR08B BB DR04 QF DR09 QF DR08A SF DR08B BB ...
      DR08B SDB DR08A QD DR08A SDB DR08B BB ...
      DR08B SF DR08A QF DR09 QF DR04 BB DR08B SDA DR08A QD ...
      DR08A SDA DR08B BB DR04 QF DR09 QF DR08A SF DR08B BB ...
      DR08B SDB DR08A QD DR08A SDB DR08B BB DR04 QF DR09 QFB DR08A ...
      SF DR08B BB DR08B SDA DR08A QDA DR07B SDA DR07A B DR06B ...
      SF DR06A QFA DR05 BB DR04A BB DR04 Q1 DR03...
      Q2 DR02 Q3 DR01 ];

NENW =  [ DR01 Q3 DR02 Q2 DR03Q1 DR04 BB DR04A BB DR05 QFA DR06A ...
      SF DR06B B DR07A SDA DR07B QDA DR08A ...
      SDA DR08B BB DR08B SF DR08A QFB DR09 ...
      QF DR04 BB DR08B SDB DR08A QD DR08A SDB DR08B BB DR08B ...
      SF DR08A QF DR09 QF DR04 BB DR08B SDA DR08A QD ...
      DR08A SDA DR08B BB DR04 QF DR09 QF DR08A SF DR08B BB ...
      DR08B SDB DR08A QD DR08A SDB DR08B BB ...
      DR08B SF DR08A QF DR09 QF DR04 BB DR08B SDA DR08A  ...
      QD DR08A SDA DR08B BB DR04 QF DR09 QF DR08A SF DR08B BB ...
      DR08B SDB DR08A QD DR08A SDB DR08B BB DR04 QF DR09 QFB DR08A ...
      SF DR08B BB DR08B SDA DR08A QDA DR07B SDA DR07A B DR06B ...
      SF DR06A QFA DR05 BB DR04A BB DR04 Q1 DR03 ...
      Q2 DR02 Q3 DR01];

ELIST =  [SWSE NENW AP];
ELIST = reverse(ELIST);
```

File spear2.m (end)

```
buildlat(ELIST);
evalin('caller','global THERING FAMLIST GLOBVAL');
disp('** Done **');
```

AT function `buildlat` builds ‘cell array’ THERING from integer indexes in `ELIST`. The `evalin('caller','global THERING FAMLIST GLOBVAL')` command ensures that lattice variables remain and are declared global in the main workspace after the execution of `spear2` stops.