MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Classical Mechanics, A Computational Approach

GETTING STARTED

The purpose of this document is to introduce a beginner to the use of the Scheme Mechanics system. This is not intended to be an introduction to the language Scheme or the notation used in the mechanics, or a catalog of the capabilities of the mechanics system.

The following instructions assume that you are using an X window system on a Unix/GNU-Linux substrate.

To Start The System

To start up the system, get an xterm shell and type

mechanics&<cr>

at the Xterm prompt. The "<cr>" appearing in the displayed line above stands for the carriage return (or enter key) on your keyboard. The ampersand "&" allows the shell to accept further commands even as the mechanics system is in use.

This will pop up two windows. One of these (the edwin-xterm window) may be minimized and you should not interact with it. The other window (Edwin) will be opened and you will interact with that window.

Edwin

Edwin is the Emacs-compatible editor, written in Scheme, that provides the primary user interface to the mechanics system.

Edwin commands use control and meta characters. Control and meta are modifiers like case shift. They are invoked by holding a modifier key depressed while typing the key to be modified. For example, the character described as C-x is typed by holding down the "control" (or "ctrl") key while typing the "x" key. While almost all keyboards have a control key, the meta key is not standard, and how it is placed depends on your xmodmap (this is magic -- ask your system wizard). It is usually mapped to the key or keys labeled "alt". Characters using the meta modifier are denoted with "M-" as in "M-f". It is common to use several modifiers simultaneously. For example, you might have a C-M-f command that is invoked by holding down both the control and the meta modifiers while typing an "f".

This document will not attempt to teach you Emacs -- you should use the Emacs tutorial that is supplied with the system to learn to use the editor effectively. It takes about 1 hour and is well worth the effort. The tutorial also explains how to evaluate scheme expressions. In the following we will assume you have gone through the tutorial. To enter the tutorial type
C-h t
at the edwin window. (That’s the character "C-h" followed by the character "t".)

To Exit The System

To exit the mechanics system, and close the editor window, type
C-x C-c
at the Edwin window. (That's "C-x" followed by "C-c".) Edwin may ask you if you want to write out any files that you may have modified, and then it will exit, closing the Scheme system.

File Naming Conventions

Edwin buffers (you will learn about them in the tutorial) have modes that determine the meanings of commands executed in them. For example, it is convenient when editing Scheme programs to have a variety of commands for balancing parentheses and for preparing programs with nice indentation. These commands would not make sense for the language C, and C editing conventions would not be very good for Scheme or text. If we follow certain conventions for naming files then the appropriate command set will be used when we edit the contents of those files. The file extension used for Scheme files is ".scm", for C it is ".c" and for text it is ".txt". For instance, the file "homework-1.scm" will be edited in scheme mode. Some commands in Scheme mode allow you to evaluate expressions, with the results appearing in the *scheme* buffer.

Working Conventions

We find it useful to construct code for a particular problem in an appropriately named file. For example, if we are writing code to study the driven pendulum we might put the code in a file called "driven-pendulum.scm". As the code is developed, expressions can be evaluated and the results appear in the *scheme* buffer. We often find it convenient to divide the edwin window and display both the code file buffer and the *scheme* buffer at the same time.

Be sure to save any changes you have made before exiting the system. To save your changes to a file put the cursor in the buffer associated with the file and type C-x C-s.

There are several ways of putting comments into a scheme file. A region can be commented by putting "#|" at the beginning of the region to be commented and "|#" at the end of the commented region. For example,

```
#|
this is a commented region,
that can extend over multiple lines
|#
```

Comments within a line are started with a semicolon ";". For example,

```
(+ 2 2)   ; same as 4
```

Comments at the begining of a line conventionally begin with ";;;". For example,

```
;;; this is code for the driven pendulum
```

An advantage of constructing the code in a separate file is that it can be reloaded (reevaluating each of the expressions in the file) in another session. To load the scheme file "double-pendulum.scm" evaluate the scheme expression: (load "double-pendulum.scm")

One can also evaluate all the expressions in a file by loading the file into an edwin buffer and typing M-o at the buffer.

It is important to comment out all regions of a file that you do not want evaluated if the file is to be loaded or if you want to be able to execute a M-o command in the associated buffer. So for example, we often write code, consisting of definitions of procedures and data. We annotate the code with the results of evaluating expressions, to help us remember how the procedures are to be called or how they work. The regions of the file that are the annotations are set off as comments. Thus the definitions can be loaded, and we can execute the demonstration examples only if we want to. For example, the following might appear in a file named "fibonacci.scm"

```
;;;----------------------------------------------------------------
;;; Fibonacci numbers

(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

#|
;;; Examples of use

(fib 20)
;Value: 6765


(iota 5)
;Value 150: (0 1 2 3 4)

(for-each (compose write-line fib)
          (iota 10))
0
1
1
2
3
5
8
13
21
34
;Unspecified return value
|#

;;; end of fibonacci.scm
;;;----------------------------------------------------------------
```

Handling Errors

You will often evaluate an improperly formed expression or an expression which has no sensible value such as "(/ 1 0)". This will cause the system to enter an error state and ask you if you want to start the debugger. If you do not want to enter the debugger answer "n".

After examining an error, whether or not you choose to enter the debugger, you should type "C-c C-c" to get back to the top level. This is important because if you don't the evaluations you subsequently perform may refer to the error state rather than to the state you intended.

Documentation

There are important sources of documentation that you should know about. The Scheme system is extensively documented using the Emacs info system. To get to the info system, type C-h i.

If at any time in a Scheme-mode buffer you want to know the possible completions of a symbol, just type C-M-i (or M-<tab>). This will pop up an Edwin window that shows all symbols known by the system beginning with your initial segment.

You may also want to know the arguments to a procedure. If the cursor is after the procedure name in an expression, you can type M-A (meta-shift-a) and a description of the arguments will appear in the Edwin command-line minibuffer (at the bottom of your Edwin window).

If you want to see the definition of any procedure (including any non-primitive system procedure) you may use the pretty printer (the pp procedure). For example:

```
(pp fib)
(named-lambda (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))))

(pp Lagrangian->Hamiltonian)
(named-lambda (Lagrangian->Hamiltonian-procedure the-Lagrangian)
  (lambda (H-state)
    (let ((t (time H-state))
          (q (coordinate H-state))
          (p (momentum H-state)))
      (define (L qdot)
        (the-Lagrangian (up t q qdot)))
      ((Legendre-transform-procedure L) p))))
```

The prinout from pp may not be exactly what you typed in, because the internal representation of your procedure may be "compiled".

You should not confuse the pretty printer (pp) with the print-expression (pe) and show-expression (se) procedures used to display the results of algebraic manipulation. These procedures do much more than print in a nicely-formatted way. They do algebraic simplification and pp or TeX the result.

You may want to recover the TeX produced by show-expression to include in a TeX document describing your results. To get this evaluate (display last-tex-string-generated).

You should not be afraid to try the debugging system. All parts of the system are self documenting. So if you enter the debugger you will see that the first line in the debugging window tells how to get out of the debugger and how to get more information about how to use it. Indeed, the *scheme* buffer itself can give you information about the Edwin commands that are relevant to it, using the command C-h m.