Symbolic Regression of $\frac{x^2}{2}$

1. GLOBAL VARIABLES

```
T = {X, ←}

(defvar x)
```

Symbolic Regression of $\frac{x^2}{2}$

2.DEFINE-TERMINAL-SET-FOR-REGRESSION

```
(defun define-terminal-set-for-
   REGRESSION ()
      (values
   '(x
     :floating-point-random-
   constant))
   )
```

3. DEFINE-FUNCTION-SET-FOR-REGRESSION

F = {+, -, *, %}

```
(defun define-function-set-for-
  REGRESSION ()
    (values '(+ - * %)
            '(2 2 2 2))
  )
```

4. PROBLEM-SPECIFIC FUNCTIONS

```
(defun %
  (numerator denominator)
    (values
  (if (= 0 denominator)
      1
      (/ numerator
  denominator)))
  )
```

5. REGRESSION-FITNESS-CASE

```
(defstruct REGRESSION-fitness-
  case
    independent-variable
    target
  )
```

6. DEFINE-FITNESS-CASES-FOR-REGRESSION

```
(defun define-fitness-cases-for-REGRESSION ()
  (let (fitness-cases x this-fitness-case)
    (setf fitness-cases (make-array *number-of-fitness-cases*))
    (format t "~%Fitness cases")
    (dotimes (index *number-of-fitness-cases*)
      (setf x (/ index *number-of-fitness-cases*))
      (setf this-fitness-case (make-REGRESSION-fitness-case))
      (setf (aref fitness-cases index) this-fitness-case)
      (setf (REGRESSION-fitness-case-independent-variable
             this-fitness-case)
              x)
      (setf (REGRESSION-fitness-case-target
             this-fitness-case)
            (* 0.5 x x))
      (format t "~% ~D      ~D      ~D"
            index
            (float x)
            (REGRESSION-fitness-case-target this-fitness-case))
    )
    (values fitness-cases)
  )
)
```

7. REGRESSION-WRAPPER

```
(defun REGRESSION-wrapper
  (result-from-program)
   (values
    result-from-program)
  )
```

Symbolic Regression of $\frac{x^2}{2}$

8. EVALUATE-STANDARDIZED-FITNESS-FOR-REGRESSION

```
(defun evaluate-standardized-fitness-for-REGRESSION
           (program fitness-cases)
  (let (raw-fitness hits standardized-fitness x target-value
        difference value-from-program this-fitness-case)
    (setf raw-fitness 0.0)
    (setf hits 0)
    (dotimes (index *number-of-fitness-cases*)
      (setf this-fitness-case (aref fitness-cases index))
      (setf x
            (REGRESSION-fitness-case-independent-variable
                   this-fitness-case))
      (setf target-value
             (REGRESSION-fitness-case-target
                   this-fitness-case))
       (setf value-from-program
             (REGRESSION-wrapper (eval program)))
       (setf difference (abs (- target-value
                                  value-from-program)))
      (incf raw-fitness difference)
      (when (< difference 0.01) (incf hits)))
    (setf standardized-fitness raw-fitness)
    (values standardized-fitness hits)))
```

9. DEFINE-PARAMETERS-FOR-REGRESSION

```
(defun define-parameters-for-REGRESSION ()
  (setf *number-of-fitness-cases* 10)
  (setf *max-depth-for-new-individuals* 6)
  (setf *max-depth-for-individuals-after-crossover* 17)
  (setf *fitness-proportionate-reproduction-fraction* 0.1)
  (setf *crossover-at-any-point-fraction* 0.2)
  (setf *crossover-at-function-point-fraction* 0.7)
  (setf *max-depth-for-new-subtrees-in-mutants* 4)
  (setf *method-of-selection* :fitness-proportionate)
  (setf *method-of-generation* :ramped-half-and-half)
  (values)
)
```

10. DEFINE-TERMINATION-CRITERION-FOR-REGRESSION

```
(defun define-termination-criterion-for-REGRESSION ;01
       (current-generation          ;02
        maximum-generations         ;03
        best-standardized-fitness   ;04
        best-hits)                  ;05
  (declare (ignore best-standardized-fitness)) ;06
  (values                           ;07
    (or (>= current-generation maximum-generations) ;08
         (>= best-hits *number-of-fitness-cases*)) ;09
  )                                 ;10
)                                   ;11
```

Symbolic Regression of $\frac{x^2}{2}$

11. REGRESSION

```
  (DEFUN                         REGRESSION                         ()
 (VALUES                       'DEFINE-FUNCTION-SET-FOR-REGRESSION
         'DEFINE-TERMINAL-SET-FOR-REGRESSION
         'DEFINE-FITNESS-CASES-FOR-REGRESSION
         'EVALUATE-STANDARDIZED-FITNESS-FOR-REGRESSION
         'DEFINE-PARAMETERS-FOR-REGRESSION
         'DEFINE-TERMINATION-CRITERION-FOR-REGRESSION
 )
)
```

12. RUN-GENETIC-PROGRAMMING-SYSTEM

```
(run-genetic-programming-system
  'REGRESSION 1.0 31 200)
```

Symbolic Regression of $\frac{x^2}{2}$

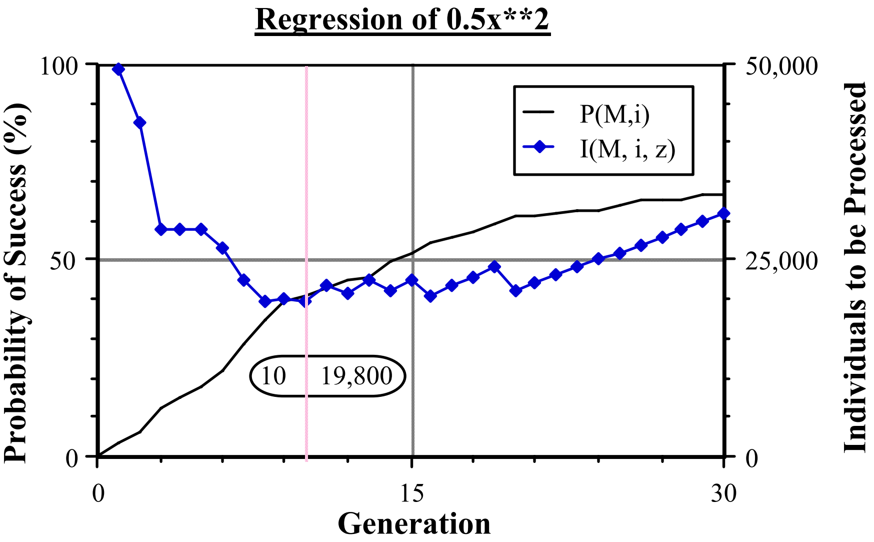
```
(run-genetic-programming-system
  'REGRESSION 1.0 1 50)
```

```
(print-population
      (run-genetic-programming-
  system 'REGRESSION 1.0 1 50))
```

```
(run-genetic-programming-system
       'REGRESSION 1.0 1 1 '(*
  0.5 x x))
```

```
(run-genetic-programming-system
  'REGRESSION 1.0 31 200)
```

Symbolic Regression of $\frac{x^2}{2}$



Performance curves based on 190 runs of the simple LISP code for the symbolic regression problem with $\frac{x^2}{2}$ as the target function and $M = 200$ and $G = 31$

Symbolic Regression of $\frac{x^2}{2}$

(1) `defvar` declaration(s),
(2) `define-terminal-set-for-REGRESSION`,
(3) `define-function-set-for-REGRESSION`,
(4) problem specific function(s),
(5) `defstruct REGRESSION-fitness-case`,
(6) `define-fitness-cases-for-REGRESSION`,
(7) `REGRESSION-wrapper`,
(8) `evaluate-standardized-fitness-for-REGRESSION`,
(9) `define-parameters-for-REGRESSION`,
(10) `define-termination-criterion-for-REGRESSION`,
(11) the function `REGRESSION`,
(12) `run-genetic-programming-system`.

Boolean Majority-On Function

1. GLOBAL VARIABLES

```
T = {d0, d1, d2}

(defvar d0)
(defvar d1)
(defvar d2)
```

2. DEFINE-TERMINAL-SET-FOR-MAJORITY-ON

```
(defun define-terminal-set-for-
  MAJORITY-ON ()
    (values '(d2 d1 d0))
  )
```

3. DEFINE-FUNCTION-SET-FOR-MAJORITY-ON

F = {AND, OR, NOT}

```
(defun define-function-set-for-
  MAJORITY-ON ()
   (values '(and or not)
           '(  2  2   1)
    )
  )
```

4. PROBLEM-SPECIFIC FUNCTIONS

5. MAJORITY-ON-FITNESS-CASE

```
(defstruct MAJORITY-ON-fitness-
  case
    d0
    d1
    d2
    target
  )
```

6. DEFINE-FITNESS-CASES

```
(defun define-fitness-cases-for-MAJORITY-ON ()
  (let (fitness-case fitness-cases index)
    (setf fitness-cases (make-array *number-of-fitness-cases*))
    (format t "~%Fitness cases")
    (setf index 0)
    (dolist (d2 '(t nil))
     (dolist (d1 '(t nil))
      (dolist (d0 '(t nil))
          (setf fitness-case
                (make-MAJORITY-ON-fitness-case)
          )
          (setf (MAJORITY-ON-fitness-case-d0 fitness-case) d0)
          (setf (MAJORITY-ON-fitness-case-d1 fitness-case) d1)
          (setf (MAJORITY-ON-fitness-case-d2 fitness-case) d2)
          (setf (MAJORITY-ON-fitness-case-target fitness-case)
                (or (and d2 d1 (not d0))
                    (and d2 (not d1) d0)
                    (or (and (not d2) d1 d0)
                        (and d2 d1 d0)
                    )
                )
          )
          (setf (aref fitness-cases index) fitness-case)
          (incf index)
          (format t
                  "~% ~D    ~S   ~S   ~S       ~S"
                  index d2 d1 d0
                  (MAJORITY-ON-fitness-case-target
                    fitness-case
                  )
          )
      )
     )
    )
    (values fitness-cases)
  )
)
```

7. MAJORITY-ON-WRAPPER

```
(defun MAJORITY-ON-wrapper
  (result-from-program)
    (values result-from-program)
  )
```

8. EVALUATE-STANDARDIZED-FITNESS-FOR-MAJORITY-ON

```
(defun evaluate-standardized-fitness-for-MAJORITY-ON
           (program fitness-cases)
  (let (raw-fitness hits standardized-fitness target-value
        match-found value-from-program fitness-case
       )
    (setf raw-fitness 0.0)
    (setf hits 0)
    (dotimes (index *number-of-fitness-cases*)
      (setf fitness-case (aref fitness-cases index))
      (setf d0 (MAJORITY-ON-fitness-case-d0 fitness-case))
      (setf d1 (MAJORITY-ON-fitness-case-d1 fitness-case))
      (setf d2 (MAJORITY-ON-fitness-case-d2 fitness-case))
      (setf target-value
            (MAJORITY-ON-fitness-case-target fitness-case))
      (setf value-from-program
            (MAJORITY-ON-wrapper (eval program)))
      (setf match-found (eq target-value value-from-program))
      (incf raw-fitness (if match-found 1.0 0.0))
      (when match-found (incf hits))
    )
    (setf standardized-fitness (- 8 raw-fitness))
    (values standardized-fitness hits)
  )
)
```

9. DEFINE-PARAMETERS-FOR-MAJORITY-ON

```
(defun define-parameters-for-MAJORITY-ON ()
  (setf *number-of-fitness-cases* 8)
  (setf *max-depth-for-new-individuals* 6)
  (setf *max-depth-for-new-subtrees-in-mutants* 4)
  (setf *max-depth-for-individuals-after-crossover* 17)
  (setf *fitness-proportionate-reproduction-fraction*
0.1)
  (setf *crossover-at-any-point-fraction* 0.2)
  (setf *crossover-at-function-point-fraction* 0.7)
  (setf *method-of-selection* :fitness-proportionate)
  (setf *method-of-generation* :ramped-half-and-half)
  (values)
)
```

10. DEFINE-TERMINATION-CRITERION-FOR-MAJORITY-ON

```
(defun define-termination-criterion-for-MAJORITY-ON
           (current-generation
            maximum-generations
            best-standardized-fitness
            best-hits)
  (declare (ignore best-standardized-fitness))
  (values (or (>= current-generation maximum-
generations)
              (>= best-hits *number-of-fitness-cases*)
          )
  )
)
```

Boolean Majority-On Function

11. MAJORITY-ON

```
(defun MAJORITY-ON ()
  (values 'define-function-set-for-MAJORITY-ON
          'define-terminal-set-for-MAJORITY-ON
          'define-fitness-cases-for-MAJORITY-ON
          'evaluate-standardized-fitness-for-MAJORITY-ON
          'define-parameters-for-MAJORITY-ON
          'define-termination-criterion-for-MAJORITY-ON
  )
)
```

12. RUN-GENETIC-PROGRAMMING-SYSTEM

```
(run-genetic-programming-system 'MAJORITY-ON 1.0 1 1
               '(or (and d2 (and d1 (not d0)))
                    (or (and d2 (and (not d1) d0))
                        (or (and (not d2) (and d1 d0))
                            (and d2 (and d1 d0))
                        )
                    )
                )
)
```

Boolean Majority-On Function

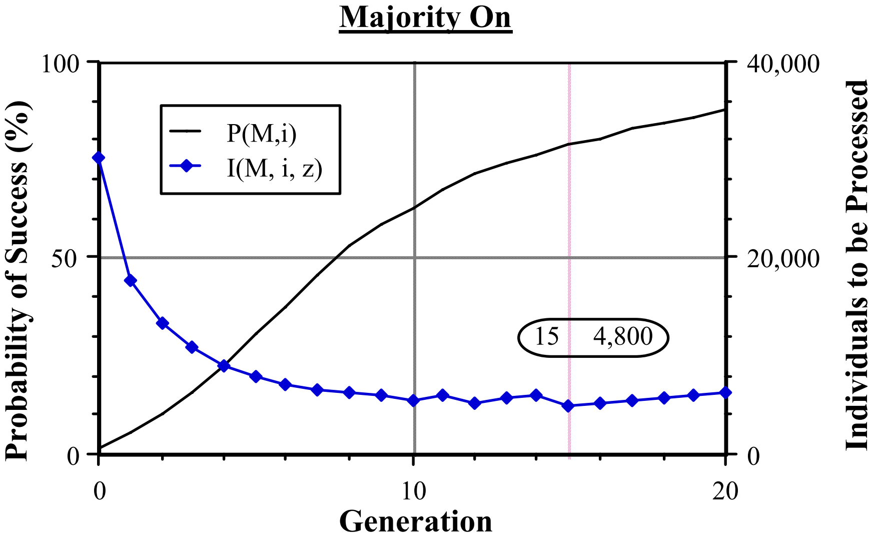


Figure B.2 Benchmark performance curves based on 330 runs of the simple LISP code for the `MAJORITY-ON` problem with a population size *M* = 100 and *G* = 21.

Discrete Non-Hamstrung Squad Car

1. GLOBAL VARIABLES

```
(defvar x)
(defvar y)
```

Discrete Non-Hamstrung Squad Car

2. DEFINE-TERMINAL-SET

```
(defun define-terminal-set-for-
  NON-HAMSTRUNG-SQUAD-CAR ()
   (values
   '((goN) (goE) (goS) (goW)))
  )
```

Discrete Non-Hamstrung Squad Car

3. DEFINE-FUNCTION-SET

F = {ifX, ifY}

```
(defun define-function-set-for-
  NON-HAMSTRUNG-SQUAD-CAR ()
    (values '(ifX ifY)
            '(  3   3)
    )
  )
```

4. PROBLEM-SPECIFIC FUNCTIONS

```
(defvar *speed-ratio* 2)


(defun goN ()
  (setf y (- y *speed-ratio*))
)

(defun goS ()
  (setf y (+ y *speed-ratio*))
)

(defun goE ()
  (setf x (- x *speed-ratio*))
)

(defun goW ()
  (setf x (+ x *speed-ratio*))
)
```

Discrete Non-Hamstrung Squad Car

MACROS

```
(ifX (goW) (goN) (goE))
```

```
#+TI (setf sys:inhibit-displacing-flag t)

(defmacro ifX (lt-0-arg eq-0-arg gt-0-arg)
  `(cond ((>= x    *speed-ratio*)  (eval ',gt-0-arg))
         ((<= x (- *speed-ratio*)) (eval ',lt-0-arg))
         (t                        (eval ',eq-0-arg))
   )
)

(defmacro ifY (lt-0-arg eq-0-arg gt-0-arg)
  `(cond ((>= y    *speed-ratio*)  (eval ',gt-0-arg))
         ((<= y (- *speed-ratio*)) (eval ',lt-0-arg))
         (t                        (eval ',eq-0-arg))
   )
)
```

Discrete Non-Hamstrung Squad Car

EVADER MACROS

```
(defmacro ifX-evader (lt-0-arg eq-0-arg gt-0-arg)
  `(cond ((>= x  1) (eval ',gt-0-arg))
         ((<= x -1) (eval ',lt-0-arg))
         (t         (eval ',eq-0-arg))
   )
)

(defmacro ifY-evader (lt-0-arg eq-0-arg gt-0-arg)
  `(cond ((>= y  1) (eval ',gt-0-arg))
         ((<= y -1) (eval ',lt-0-arg))
         (t         (eval ',eq-0-arg))
   )
)

(defun goN-evader ()
  (setf y (+ y 1))
)

(defun goS-evader ()
  (setf y (- y 1))
)

(defun goE-evader ()
  (setf x (+ x 1))
)

(defun goW-evader ()
  (setf x (- x 1))
)
```

5. FITNESS-CASE

```
(defstruct NON-HAMSTRUNG-SQUAD-
  CAR-fitness-case
    x
    y
  )
```

Discrete Non-Hamstrung Squad Car

6. DEFINE-FITNESS-CASES

```
(defun define-fitness-cases-for-NON-HAMSTRUNG-SQUAD-CAR ()
  (let (fitness-case fitness-cases index)
    (setf fitness-cases (make-array *number-of-fitness-cases*))
    (format t "~%Fitness cases")
    (setf index 0)
    (dolist (x '(-5 5))
      (dolist (y '(-5 5))
          (setf fitness-case
                (make-NON-HAMSTRUNG-SQUAD-CAR-fitness-case)
          )
          (setf (NON-HAMSTRUNG-SQUAD-CAR-fitness-case-x
                  fitness-case
                )
                x
          )
          (setf (NON-HAMSTRUNG-SQUAD-CAR-fitness-case-y
                  fitness-case
                )
                y
          )
          (setf (aref fitness-cases index) fitness-case)
          (incf index)
          (format t "~% ~D    ~S   ~S" index x y)
      )
    )
    (values fitness-cases)
  )
)
```

7. WRAPPER

```
(defun NON-HAMSTRUNG-SQUAD-CAR-
  wrapper (argument)
    (values argument)
  )
```

Discrete Non-Hamstrung Squad Car

8. EVALUATE-STANDARDIZED-FITNESS

```
(defun evaluate-standardized-fitness-for-NON-HAMSTRUNG-SQUAD-CAR
           (program fitness-cases)
  (let (raw-fitness hits standardized-fitness
        e-delta-x e-delta-y p-delta-x p-delta-y
        time-tally old-x old-y
        criterion
        (number-of-time-steps 50)
       )
    (setf criterion *speed-ratio*)
    (setf raw-fitness 0.0)
    (setf hits 0)
    (dotimes (icase *number-of-fitness-cases*)
      (setf x (NON-HAMSTRUNG-SQUAD-CAR-fitness-case-x
                (aref fitness-cases icase)
              )
      )
      (setf y (NON-HAMSTRUNG-SQUAD-CAR-fitness-case-y
                (aref fitness-cases icase)
              )
      )
      (setf time-tally 0.0)
```

Discrete Non-Hamstrung Squad Car

```
      (catch :terminate-fitness-case-simulation
        (dotimes (istep number-of-time-steps)
          (setf old-x x)
          (setf old-y y)
          (when (and (<= (abs x) criterion)
                     (<= (abs y) criterion)
                )
            (incf hits)
            (throw :terminate-fitness-case-simulation
                   :scored-a-hit
            )
          )
          ;; Note: (x,y) is position of the Evader.
          ;; Changing the position of EVADER changes X and Y.
          ;; Execute evader player for this time step
          (eval '(ifY-evader
                   (goS-evader)
                   (ifX-evader (goW-evader)
                               (goS-evader) (goE-evader)
                   )
                   (goN-evader)
                 )
          )
          (setf e-delta-x (- old-x x))
          (setf e-delta-y (- old-y y))
          ;; Reset position for Pursuer player.
          (setf x old-x)
          (setf y old-y)
          (NON-HAMSTRUNG-SQUAD-CAR-wrapper (eval program))
          (setf p-delta-x (- old-x x))
          (setf p-delta-y (- old-y y))
          ;; Integrate x and y changes.
          (setf x (- old-x (+ p-delta-x e-delta-x)))
          (setf y (- old-y (+ p-delta-y e-delta-y)))
          (incf time-tally)
        )
      )
      (incf raw-fitness time-tally)
    )
    (setf standardized-fitness raw-fitness)
    (values standardized-fitness hits)
  )
)
```

Discrete Non-Hamstrung Squad Car

9. DEFINE-PARAMETERS-FOR-NON-HAMSTRUNG-SQUAD-CAR

```
(defun define-parameters-for-NON-HAMSTRUNG-SQUAD-CAR ()
  (setf *number-of-fitness-cases* 4)
  (setf *max-depth-for-new-individuals* 6)
  (setf *max-depth-for-new-subtrees-in-mutants* 4)
  (setf *max-depth-for-individuals-after-crossover* 17)
  (setf *fitness-proportionate-reproduction-fraction* 0.1)
  (setf *crossover-at-any-point-fraction* 0.2)
  (setf *crossover-at-function-point-fraction* 0.7)
  (setf *method-of-selection* :fitness-proportionate)
  (setf *method-of-generation* :ramped-half-and-half)
  (values)
)
```

10. DEFINE-TERMINATION-CRITERION

```
(defun define-termination-criterion-for-NON-HAMSTRUNG-SQUAD-CAR
           (current-generation
            maximum-generations
            best-standardized-fitness
            best-hits)
  (declare (ignore best-hits best-standardized-fitness))
  (values (>= current-generation maximum-generations))
)
```

11. NON-HAMSTRUNG-SQUAD-CAR

```
(defun NON-HAMSTRUNG-SQUAD-CAR ()
  (values
     'define-function-set-for-NON-HAMSTRUNG-SQUAD-CAR
     'define-terminal-set-for-NON-HAMSTRUNG-SQUAD-CAR
     'define-fitness-cases-for-NON-HAMSTRUNG-SQUAD-CAR
     'evaluate-standardized-fitness-for-NON-HAMSTRUNG-SQUAD-CAR
     'define-parameters-for-NON-HAMSTRUNG-SQUAD-CAR
     'define-termination-criterion-for-NON-HAMSTRUNG-SQUAD-CAR
  )
)
```

12. RUN-GENETIC-PROGRAMMING-SYSTEM

```
(run-genetic-programming-system
     'NON-HAMSTRUNG-SQUAD-CAR
  1.0 21 100
  )
```

```
(run-genetic-programming-system
  'NON-HAMSTRUNG-SQUAD-CAR 1.0 1
  1
    '(ifX (goW) (ifY (goS) (goS)
  (goN)) (goE))
  )
```