NISTIR 4437

A PROGRAM FOR CALCULATING THE MAXIMUM RADIATION ON A WALL

Richard L. Smith

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Engineering Laboratory
Center for Fire Research
Gaithersburg, MD 20899

Supported in part by:
U.S. Air Force Engineering and Services Center, Airbase Fire Protection and Crash Rescue System Branch
Tyndall, AFB, FL 32403-6001

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4437

A PROGRAM FOR CALCULATING THE MAXIMUM RADIATION ON A WALL

Richard L. Smith

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Engineering Laboratory
Center for Fire Research
Gaithersburg, MD 20899

Supported in part by:
U.S. Air Force Engineering and Services Center, Airbase Fire Protection and Crash Rescue System Branch
Tyndall, AFB, FL 32403-6001

October 1990

TABLE OF CONTENTS

Abstract

This report describes a program module of the expert system EXPOSURE. This module, written in Common Lisp, is for calculating the maximum electromagnetic radiation incident upon a building's wall due to the burning of a neighboring building. It is assumed that the burning building has an arbitrary number of rectangular openings emitting radiation. The exposed wall can be considered as being composed of a number of rectangular regions. These regions may be openings or regions of different materials. This program can determine the maximum radiation for each of these regions and for the wall less these regions.

Keywords: exposure, radiation, radiative heat transfer, computer programs.

Forward

In this report we are trying a somewhat unusual approach to documenting a computer program. This entire report is the contents of the computer file that contains the program "Radiation." The only things omitted are some semicolons that indicate that the written texts of this report are comments and the initial header strip:

```
;; -*- Mode: Joshua; Package: FIRE-PROTECTION; Syntax: Joshua; Default-character-style:
;; (:FIX :ROMAN :LARGE) -*- Created 7/27/89 16:15:45 by Smith running on SMOKEY at
;; NBS.
```

The only addition to this report that is not in the file is the figure. A computer program is a dynamic thing and should be documentated during development as well as when it is finished. This report was generated by printing a file which is a module in an expert system. It represents a snap shot of the module at one instance of time. Since the computer code is the focus of this report it is included in the body of the report. The document is designed to be read in its entirety although the audience wanting to do so is probably small. We believe this type of documentation is required for modules of expert systems which are going to be used to make real world recommendations. This type of document will allow others to review, revise, and maintain this code and its physical science basis. Those who do not understand LISP code can skip reading the code. which is printed in bold type style, enclosed in parentheses, and often starts with **def**.... These readers should be able to follow the logic of the program but they may not be able to determine if the code performs as described.

1. Introduction

This report documents a module of the expert system EXPOSURE. It describes a program for calculating the maximum electromagnetic radiation incident upon a building's wall due to the burning of a neighboring building. It is assumed that the burning (exposing) building has an arbitrary number of rectangular openings emitting radiation. The exposed wall can be considered as being composed of a number of rectangular regions. These regions may be openings or regions of different materials. We wish to determine the maximum radiation for each of these regions and for the wall less these regions.

After rescuing people from a burning building, the next highest priority item for fire fighters is to prevent the fire from spreading to adjacent buildings [1]. Because of the potential damage that can be caused if fire propagates from building to building there are standards that provide guidance on how to reduce the likelihood of fire spreading [2] [3].

A critical part of the analysis of the spread of fire between buildings is the calculation of the electromagnetic radiation falling upon the exposed surfaces of the nonburning building. The calculations of the radiation is complex and does not lend itself to simple approximations [5] [6] [7] [8]. Present standards are based on work done before the advent of the personal computer [4] [5]. To make the estimating of the radiation level in the standards possible, a number of simplifying assumptions had to be made in the existing standards.

NFPA 80A, Recommended Practice for Protection of Buildings from Exterior Fire Exposures [2], makes a number of assumptions. Some are used in this expert system others are not. We do make the following assumptions which are also made by NFPA 80A:

1. Only one wall of the burning building will expose one wall of the exposed building.

2. All radiating openings are assumed to have a rectangular shape [5].

We do not make the following assumptions that are made by the NFPA 80A.

1. The exposed and exposing walls are parallel (for an exception see [3]).

2. A region of a wall with a number of radiators can be treated as a single radiator with a size of the rectangle which encloses all the individual openings and which radiates at a reduced intensity. The reduced intensity equals the ratio of the area of the openings to that of the enclosing rectangle.

3. If the exposed building has a combustible wall, then the minimum allowed separation between buildings is the separation such that the thermal radiation falling on the exposed building just equals 1.26 w./sq. cm. (the piloted ignition level for oven dried wood).

4. Only the maximum radiation level on the exposed wall is determined.

Briefly, this program does the following: Based upon the size of an opening and its minimum distance from the exposed wall, the program divides the opening into subelements. It then finds the configuration factor due to one opening by summing the contributions from all the small elements of the area in the opening. It then sums all the contributions from all the openings. This gives the configuration factor at any point on the exposed wall due to all the openings. The program then can determine the maximum configuration factor in any rectangular region on the exposed wall by scanning the region. Finally, it has the capability of finding the maximum configuration on the area of the exposed wall less its openings.

The program described in this report was written in Symbolics Common LISP. (Note: Certain commercial products are identified in this report in order to adequately specify the computer program. Such identification does not imply recommendation by the National Institute of Standards and Technology, nor does it imply that these products identified are necessarily the best available for the purpose.)

2. Source Flux

We assume all surfaces of openings in the exposing building radiate according to the equation

$$E = \varepsilon\lambda T\text{^}4 \text{ w/sq cm.} \qquad (1)$$

where ε is the emissivity ($\varepsilon \leq 1$). For an ideal black body $\varepsilon = 1$ and where λ is the Stefan Boltzmann constant with a value of 5.6703e-12 (w./sq. cm.)(K)^-4. We will assume $\varepsilon = 1$ for all the openings. Since a real opening will always radiate less than a black body at the same temperature, this assumption results in overestimating the radiation and therefore errs on the side of safety.

We record the Stefan Boltzmann constant in the program as follows:

```
(defconstant *Stefan-Boltzmann-constant* 5.6703e-12 "(w/sq cm)(K)^-4")
```

The LISP function **black-body-flux** is the LISP version of eq. 1. It takes as its input or its parameter the temperature of the burning compartment. It returns the radiation flux at the surface of the openings of the building in w./sq. cm.

```
(defun black-body-flux (temperature)        ; Temperature in Kelvin
 (* *Stefan-Boltzmann-constant* (expt temperature 4)))
```

3. Configuration Factor

If we assume Lambert's cosine law holds, the incident radiant flux falling on a small element of surface is

$$Rf = E \int (\cos \alpha 1 \ \cos \alpha 2)/(\pi \ r\text{^}2)\ dA1 \qquad (2)$$

where we integrate over the surface *A1*, and *E* is the source flux determined above. The integral is called the configuration factor. From Figure 1. we see that *r* is the distance between the point of integration at *dA1* to the point at which we are evaluating the field, $\alpha 1$ is the angle between the normal to *dA1* (the exposing surface) and *r*, and $\alpha 2$ is the angle between the normal to the exposed surface at the point we are evaluating the field and *r*. If *(xa, ya, za)* is the source point on *dA1* and *(x0, y0, z0)* is the field point on the exposed surface, then we have

$$r = ((x0 - xa)\text{^}2 + (y0 - ya)\text{^}2 + (z0 - za)\text{^}2)\text{^}.5$$

If we define γ and $\beta 1$ such that

$$\cos\gamma = (x0 - xa)/r$$

$$\sin\gamma = (y0 - ya)/r$$

$$\cos\beta 1 = (x2 - x1)/L1$$

$$\sin\beta 1 = (y2 - y1)/L1$$

where $\beta 1$ is the slope of wall1 (the exposing wall) and the length of the wall is given by

$$L1 = ((x2 - x1)\text{^}2 + (y2 - y1)\text{^}2)\text{^}.5$$

then we have

$$\cos \alpha 1 = \cos\gamma \ \sin\beta 1 - \sin\gamma \ \cos\beta 1.$$

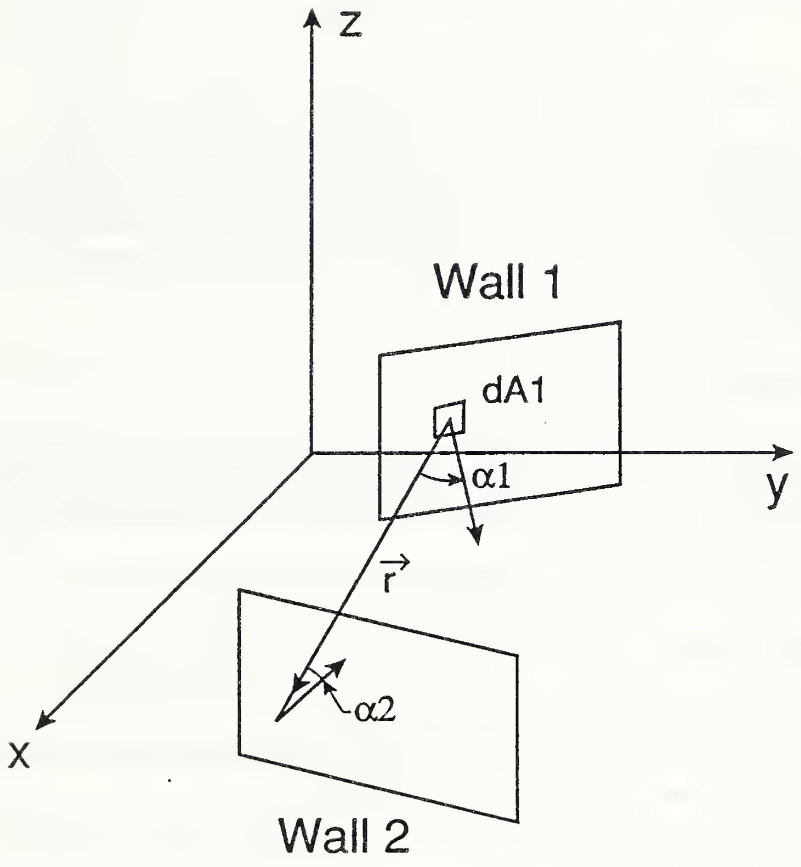

Figure 1.0 Variables of the Configuration Integral

If we define $\beta 2$ such that

$$\cos\beta 2 = (x4 - x3)/L2$$

$$\sin\beta 2 = (y4 - y3)/L2$$

where $\beta 2$ is the slope of wall2 and the length of exposed wall (wall2) is given by

$$L2 = ((x4 - x3)^2 + (y4 - y3)^2)^{.5}$$

then we have

$$\cos \alpha 2 = \cos\gamma \sin\beta 2 - \sin\gamma \cos\beta 2.$$

All rectangles are defined by giving the coordinates of the lower left hand corner and the upper right hand corner. This is sufficient to define a rectangle because all rectangles of interest are either vertical or horizontal. Therefore, an opening or a wall can be defined by the four points: *(x1, y1, z1)*, *(x2, y2, z1)*, *(x2, y2, z2)*, and *(x1, y1, z2)* or by using the two point convention, *(x1, y1, z1)*
and *(x2, y2, z2)*.

3.1 Program Input and Basic Functions

The input for this program is the lower left and upper right corners of the exposing wall and its openings and the exposed wall and its openings. Important concepts for this program are various types of points and rectangles. In this section we give their definitions and the basic functions for operating on them.

One basic entity is the point which we define as:

```
(defstruct (point :alterant)
 x y z cf)
```

Sometimes we have an interest in the configuration factor for a point so it has been included.

Another basic entity is the vertical rectangle which we define as:

```
(defstruct (vertical-rectangle)
 lower-left upper-right)
```

Then walls, openings, and elements of integration are just instances of vertical rectangles.

```
(defstruct (wall (:include vertical-rectangle)))

(defstruct (opening (:include vertical-rectangle)))

(defstruct (element :alterant (:include vertical-rectangle)))
```

We will have use for two simple functions. One is the `square-root` and the other is the `square`. We define these functions as follows:

```
(defsubst square-root (number)
 (sqrt number))

(defsubst square (number)
 (expt number 2))
```

We used `defsubst` so they get expanded at compile time.

We will have use for a function which rounds off some of our results to a controllable number of significant figures. We call that function `round-off`.

```
(defun round-off (number &optional (significant-figures 0))
  (let ((d (expt 10 significant-figures)))
 (/ (round (* number d)) d)))
```

We will have need to determine the separation between two points.

```
(defun point-separation (point-1 point-2) ;3 dimensions
 (square-root
  (+ (square (- (point-x point-2) (point-x point-1)))
     (square (- (point-y point-2) (point-y point-1)))
     (square (- (point-z point-2) (point-z point-1))))))
```

We use `point-separation` to determine the length of the diagonal of vertical rectangles.

```
(defun vertical-rectangle-diagonal (vertical-rectangle)
  (point-separation (vertical-rectangle-lower-left vertical-rectangle)
                    (vertical-rectangle-upper-right vertical-rectangle)))
```

We will also need the separation of the projection of two points on the xy-plane.

```
(defun xy-separation (point-1 point-2)      ;separation in xy plane
 (square-root
  (+ (square (- (point-x point-2) (point-x point-1)))
     (square (- (point-y point-2) (point-y point-1))))))
```

We use `xy-separation` to determine the width of vertical rectangle.

```
(defun vertical-rectangle-width (vertical-rectangle)
  (xy-separation (vertical-rectangle-lower-left vertical-rectangle)
                 (vertical-rectangle-upper-right vertical-rectangle)))
```

Instead of calling the wall dimension in the xy plane its width, we called it length.

```
(defun wall-length (wall)
  (vertical-rectangle-width wall))          ;note switch form length to width
```

But for an opening we switch back to width.

```
(defun opening-width (opening)
 (vertical-rectangle-width opening))
```

We will need the height of our rectangles so we define the following:

```
(defun vertical-rectangle-height (vertical-rectangle)
  (- (point-z (vertical-rectangle-upper-right vertical-rectangle))
     (point-z (vertical-rectangle-lower-left vertical-rectangle))))

(defun wall-height (wall)
 (vertical-rectangle-height wall))

(defun opening-height (opening)
 (vertical-rectangle-height opening))
```

Likewise, we will need the area of openings and elements of integration.

```
(defun vertical-rectangle-area (vertical-rectangle)
 (* (vertical-rectangle-height vertical-rectangle)
    (vertical-rectangle-width vertical-rectangle)))
```

```
(defun opening-area (opening)
 (vertical-rectangle-area opening))

(defun element-area (element)
 (vertical-rectangle-area element))
```

The following functions are used to calculate the coordinates of the centers of various rectangles.

```
(defun vertical-rectangle-center (vertical-rectangle)
 (let ((ll (vertical-rectangle-lower-left vertical-rectangle))
       (ur (vertical-rectangle-upper-right vertical-rectangle)))
   (make-point :x (/ (+ (point-x ur) (point-x ll)) 2)
               :y (/ (+ (point-y ur) (point-y ll)) 2)
               :z (/ (+ (point-z ur) (point-z ll)) 2))))


(defun opening-center (opening)
 (vertical-rectangle-center opening))

(defun element-center (element)
 (vertical-rectangle-center element))
```

The slope of the various rectangles in the xy-plane can be determined by these functions.

```
(defun slope-of-vertical-rectangle (vertical-rectangle)
 (let* ((ll (vertical-rectangle-lower-left vertical-rectangle))
        (ur (vertical-rectangle-upper-right vertical-rectangle))
        (y2 (point-y ur))
        (y1 (point-y ll))
        (x2 (point-x ur))
        (x1 (point-x ll)))
   (atan (- y2 y1) (- x2 x1))))

(defun slope-of-wall (wall)
 (slope-of-vertical-rectangle wall))

(defun slope-of-opening (opening)
 (slope-of-vertical-rectangle opening))

(defun slope-of-element (element)
 (slope-of-vertical-rectangle element))
```

This function determines the cosines of the alpha angles.

```
(defun cos-alphas (element field-point field-wall)
  (let* ((el-center (element-center element))
         (r (point-separation el-center field-point))
         (cosγ (/ (- (point-x field-point)
                     (point-x el-center)) r))
         (sinγ (/ (- (point-y field-point)
                     (point-y el-center)) r))
         (β1 (slope-of-element element))
         (β2 (slope-of-wall field-wall))
         (cos-α1 (- (* cosγ (sin β1))
                    (* sinγ (cos β1))))
         (cos-α2 (- (* cosγ (sin β2))
                    (* sinγ (cos β2)))))
    (values (abs cos-α1) (abs cos-α2))))
```

3.2 Configuration Factor of an Element

Whenever we have the value of the diagonal, *d*, of an element of an opening much less than the value of the shortest value of *r* over the opening, i.e., *d << r-min* where

```
d = ((x2 - x1)^2 + (y2 - y1)^2 + (z2 - z1)^2)^.5
```

then *α1, α2,* and *r* do not vary appreciably over the element of area *A1*.
Therefore, we can approximate the configuration integral for one element by the following expression:

```
cf = (cos α1 cos α2)A1/(pi r^2)
```

We will define **pi** to be **3.141593**. This avoids double precision calculations and thereby speeds up the computations.

```
(defvar *single-p-pi* 3.141593)
```

We now define the LISP function **configuration-factor-for-one-element** with parameters of **element**, **field-point**, and **field-wall**. It returns the numerical value for the configuration factor for this infinitesimal element at the **field-point**.

```
(defun configuration-factor-for-one-element (element field-point field-wall)
  (multiple-value-bind (cosα1 cosα2)
      (cos-alphas element field-point field-wall)
  (/ (* cosα1 cosα2)
        (element-area element))
    (* (square (point-separation (element-center element) field-point))
       *single-p-pi*))))
```

We have found that if we have *d ≤ .1 * r-min*, we do not need to integrate over the area with this value of d. We get a good approximation for our integral by using **configuration-factor-for-one-element**. Therefore we define our constant ***limit-element-diagonal-to-r-min-ratio*** as follows

```
(defvar *limit-element-diagonal-to-r-min-ratio* .1)
```

If our area of interest does not satisfy this condition, we must subdivide it into subelements until it does.

To implement this approximation we need to be able to 1.) calculate the shortest distance between a point and a planar region *(r-min)* and 2.) determine the number of subelements so we can use the above approximation to calculate the configuration factor.

3.3 Shortest Distance Point to Planar Region

We wish to determine the shortest distance between a point *(x0 y0 z0)* and a planar rectangular region defined by its coordinates of opposite corners *(x1 y1 z1)* and *(x2 y2 z2)*. We assume the region lies in a plane that is perpendicular to the xy plane, i.e. the vertical plane.

The inputs for the function **shortest-distance-point-to-plane-region** are the **point** and **vertical-rectangle**. It returns the shortest distance and the coordinates of the nearest point in the rectangle.

```
(defun shortest-distance-point-to-plane-region
        ;;for vertical rectangle in rectangular world
        (point vertical-rectangle)
  (let* ((xi 0)                                    ;xi etc. will be the nearest point in the region.
         (yi 0)                                    ;Value of 0 for xi, yi, & zi are to avoid nil.
         (zi 0)
         (zll (point-z
                (vertical-rectangle-lower-left vertical-rectangle)))
         (zu (point-z
               (vertical-rectangle-upper-right vertical-rectangle)))
         (y1 (point-y
               (vertical-rectangle-lower-left vertical-rectangle)))
         (y2 (point-y
               (vertical-rectangle-upper-right vertical-rectangle)))
         (yll (min y1 y2))
         (yu (max y1 y2))
         (x1 (point-x
               (vertical-rectangle-lower-left vertical-rectangle)))
         (x2 (point-x
               (vertical-rectangle-upper-right vertical-rectangle)))
         (xll (min x1 x2))
         (xu (max x1 x2))
         (y0 (point-y point))
         (x0 (point-x point)))
  ;; This condition on z works only for vertical walls
   (setq zi (min zu (max zll (point-z point))))
  ;; The x and y coordinates of a point in the planar region are not independent.
  ;; They are related by the equation y = mx + b where m is the slope given
  ;; by (y2 - y1)/(x2 - x1). We need to deal with two special cases which are
  ;; when m = 0 and 1/m = 0.
  (cond ((= x2 x1)                                 ;Vertical slope= ∞ because xi = x1.
          (setq xi x1)                             ;All x's have same value on plane of region.
          (setq yi (min yu (max yll y0))))
         ((= y2 y1)                                ;m = 0 so 1/m = ∞
          (setq yi y1)
          (setq xi (min xu (max xll x0))))
         ;;Now we deal with the general case.
         (t (let* ((m (/ (- y2 y1)(- x2 x1)))   ;Slope of opening in xy plane.
                   ;; Substituting y1 for y and x1 for x and solving for b we have
                   (b (- y1 (* m x1)))
                   ;;The equation for the line which is
                   ;;perpendicular to above line is y = (1/m)x+ b1.
                   ;;Since the point (x0, y0) lies on this line
                   ;; we have
                   (b1 (+ y0 (/ x0 m)))
                   ;;The intersection of these two lines is given by
                   (xii (/ (- b1 b) (+ m (/ 1 m))))
                   (yii (/ (+ b (* m m b1)) (+ (* m m) 1))))
             (cond ((and (or (< xll xii) (= xll xii))
                          ;;The intersection is in the region
                          (or (< xii xu) (= xu xii)))
                     (setq xi xii)(setq yi yii))
                    ((< xii xll)(setq xi xll)(setq yi (+ (* m xi) b)))
                    ((> xii xu)(setq xi xu) (setq yi (+ (* m xi) b)))))))
  (list
   (square-root (+ (square (- x0 xi)) (square (- y0 yi))
                   (square (- (point-z point) zi))))
   (make-point :x xi :y yi :z zi))))   ;nearest point
```

3.4 Number of Subelements in Opening

We have an opening with sides **longer**, *L*, and **shorter**, *s*, we want the diagonal to be less than ***limit-element-diagonal-to-r-min-ratio*** times the minimum value of the distance from the field point and any point in the opening. If

*((L^2 + s^2)^.5) < (****limit-element-diagonal-to-r-min-ratio****)*r-min,*

then we are done. However, if this is not the case, we need to subdivide the opening into approximate square elements. Therefore, we let *n = L/s*, then if we divide *L* by n we get square elements. If we divide *L* by *N = (round n)* we get nearly square elements. If

*((L/n)^2 + s^2)^.5 < (****limit-element-diagonal-to-r-min-ratio****)*r-min,*

we are done. Otherwise, we must continue making our elements smaller by reducing the size of both dimensions by dividing them by *p*.

Then we want an integer *P* such that

(((L/N)^2 + s^2)^.5)/P <
*(****limit-element-diagonal-to-r-min-ratio****)*r-min < (((L/N)^2 +*
s^2)^.5)/(P -1).

This can be achieved as long as *r-min* is not zero. If *r-min* is small, *P* may become so large as to make this approach undesirable.

We make

*p = (((L/N)^2 + s^2)^.5)/(****limit-element-diagonal-to-r-min-ratio****)*r-min.*

It follows that *P =* **(ceiling p)**. We note that **(ceiling p)** is a LISP expression which returns an integer which equals *p* if *p* is an integer otherwise it returns the first integer larger than *p*.

We recall that LISP can't tell the different between *N* and *n*.

The function **subelements-parameters** takes the width and height of the opening and **r-min** as input and returns a list which has the form:
(divisor-of-width divisor-of-height).

```
(defun subelements-parameters (opening-width opening-height r-min)
 (cond ((= r-min 0)
        (error
         "I can not compute the solution for two touching walls,
  i.e., r-min = 0."))
       (t (let* ((longer (max opening-width opening-height ))
                 (shorter (min opening-width opening-height))
                 (N (round (/ longer shorter)))
                 (P (ceiling
                     (/ (square-root (+ (square (/ longer N))
                                        (square shorter)))
                        (* *limit-element-diagonal-to-r-min-ratio* r-min)))))
            (cond ((> opening-width opening-height)
                   (values (* P N) P))      ;divisor-of-width divisor-of-height
                  (t (values P (* P N)))))))))
```

3.5 Configuration Factor for One Opening

To obtain the configuration factor for one opening, we sum the configuration factor for all the subelements in that opening. The function **configuration-factor-for-one-opening** take as input the **field-point**, **opening**, and **exposed-wall**. It returns the configuration factor of the opening at the **field-point**.

```
(defun configuration-factor-for-one-opening ;on exposed wall at field point
       (field-point opening exposed-wall)
   (let ((width-of-opening (opening-width opening))
         (height-of-opening (opening-height opening))
         (β1 (slope-of-opening opening))
         ;; rm is the shortest distance from the opening to the field point.
         (rm (car
               (shortest-distance-point-to-plane-region
                 field-point opening))))
     (multiple-value-bind (divisor-of-width divisor-of-height)
         (subelements-parameters width-of-opening height-of-opening rm)
       (let* ((element-length (/ width-of-opening divisor-of-width))
              (element-height
                (/ height-of-opening divisor-of-height))
              ;;height-of-opening = (- z2 z1)
              (x1 (point-x (opening-lower-left opening)))
              (y1 (point-y (opening-lower-left opening)))
              (z1 (point-z (opening-lower-left opening)))
              (lower-left-element-point
                (make-point :x x1 :y y1 :z z1))
              (upper-right-element-point
                (make-point :x (+ x1 (* element-length (cos β1)))
                            :y (+ y1 (* element-length (sin β1)))
                            :z (+ z1 (* element-height))))
              (scanning-element (make-element :lower-left lower-left-element-point
                                              :upper-right upper-right-element-poin
t)))
         (loop for i from 0 to (- divisor-of-height 1)
              ;;loop over elements in opening
              with sum = 0
              do
           (alter-point lower-left-element-point z (+ z1 (* i element-height)))
           (alter-point upper-right-element-point z (+ z1 (* (+ i 1) element-height)))
          (loop for j from 0 to (- divisor-of-width 1)
                do
            (alter-point lower-left-element-point x (+ x1 (* j element-length (cos β1))
)
                         y (+ y1 (* j element-length (sin β1))))
            (alter-point upper-right-element-point x (+ x1 (* (+ j 1) element-length (c
os β1)))
                         y (+ y1 (* (+ j 1) element-length (sin β1))))
            (alter-element scanning-element lower-left lower-left-element-point
                           upper-right upper-right-element-point)
           (setq sum (+ sum
                        (configuration-factor-for-one-element
                          scanning-element field-point exposed-wall))))
              finally (return sum))))))
```

Comparing the results of **configuration-factor-for-one-opening** with analytical expressions for three special cases showed agreement to 4 or 5 decimal places (see Appendix A). Note that this is an approximate integration which depends upon such things as the value of ***limit-element-diagonal-to-r-min-ratio***. The exactness of this approximation can be improved by the proper adjustment of this parameter.

3.6 Configuration Factor for N Openings

If we have an arbitrary number of openings, we sum the configuration factors for all the individual openings. The function `configuration-factor-for-n-openings` has input of the `field-point`, `list-of-openings`, and `exposed-wall` It returns the configuration factor due to all the openings at the field point.

```
(defun configuration-factor-for-n-openings
       (field-point list-of-openings exposed-wall)
 (loop for opening in list-of-openings
       summing
            (configuration-factor-for-one-opening
             field-point opening exposed-wall)))
```

At this point we can calculate the field at any point on the exposed wall.

4. Determining Location and Strength of Maximum of an Opening on an Exposed Wall

The next step is to determine the locations and strengths of the maximum configuration factor (or field) on an exposed wall due to the radiation from one opening in the exposing wall.

If we have a group of openings for which the separation between openings is much smaller than the shortest distance to the exposed wall, then such a block of openings can be treated as a single opening whose area is the sum of the individual opening's areas.

It can be shown that the maximum for an element of an opening will lie on the exposed wall at the point closest to the center of the element.

To find the the maximum of an opening we will start by guessing its location. Then we will search for a point that has a configuration factor that is close to the maximum configuration factor. Close means within 2% or less. So we need to pick a point for our initial guess and the size of steps we will take while searching for our maximum.

Because of symmetry in the *z* direction, the maximum for an opening will always have a *z* value equal to the midpoint height of the opening if it falls in the exposed rectangle of interest. In addition, the maximum of an opening frequently lies at the point which is nearest the center of the opening. This tends to be a better guess the smaller the alpha angle. However, it is still reasonable even out to 45 degrees. Our first guess for the location for the maximum is at the point nearest the center of the opening. Then we try to walk to a greater value for the configuration factor. We move in a plane parallel to the *xy* plane a distance given by the step size and see what the configuration factor is at that point.

The function `step-size` takes as input the `opening` and the `exposed-wall`. To estimate the `step-size` we assume that the relative change in the configuration factor goes as *(*`step-size`*/separation)*^2 which is characteristic of an inverse square relationship where the change is approximately perpenticular to *r*. We believe this is a conservative assumption. Significant improvement in
the code's speed may be achieved by an improvement here.

```
(defun step-size (opening exposed-wall)
  (let* ((f .02)                               ;max fractional change in cf 2%
         (source-point (opening-center opening))
         (r (first (shortest-distance-point-to-plane-region
                     source-point exposed-wall))))
   (* r (square-root f))))
```

The function `Maximum-of-an-Opening` takes as input the `opening` and `exposed-wall` and returns the `field-point` (the location of the maximum for the opening) with the `cf` slot having a value of `cfmo` (the maximum configuration factor for the opening).

```
(defun Maximum-of-an-Opening                     ;returns the max-field point
       (opening exposed-wall)
 (let* ((center-of-opening (opening-center opening))
        (short1 (shortest-distance-point-to-plane-region
                  center-of-opening
                  exposed-wall))
        (field-point (second short1))   ;nearest point
        (β2 (slope-of-wall exposed-wall))
        (step (step-size opening exposed-wall))
        (cfmo
         (configuration-factor-for-one-opening
          field-point opening exposed-wall))
        (new-field-point+ (make-new-field-point-positive-step
                            exposed-wall field-point step β2))
        (new-field-point- (make-new-field-point-negative-step
                            exposed-wall field-point step β2))
        (cf+ (configuration-factor-for-one-opening
              new-field-point+ opening exposed-wall))
        (cf- (configuration-factor-for-one-opening
              new-field-point- opening exposed-wall)))
  (cond ((> cf+ (+ cfmo .00001))
         (search-for-max-plus
          exposed-wall new-field-point+ step β2 cfmo opening))
        ((> cf- (- cfmo .00001))                 ;This and cf+ make the following
        ;;approximate equal to.
         (search-for-max-minus
          exposed-wall new-field-point- step β2 cfmo opening))
        (t (alter-point field-point              ;We add info to the field point.
                        cf cfmo)
          field-point))))                        ;No larger cf, return altered field point.
```

We first try to move in a positive x or y direction. If we don't find a maximum, we look in the negative direction. We note that for one opening the configuration factor is monotonic to its maximum. We take a step by making a new field point either in the positive or negative direction. We first write the function for a positive step.

```
(defun make-new-field-point-positive-step
       (exposed-wall field-point step β2)
 (let ((x3 (point-x (wall-lower-left exposed-wall)))
       (x4 (point-x (wall-upper-right exposed-wall)))
       (y3 (point-y (wall-lower-left exposed-wall)))
       (y4 (point-y (wall-upper-right exposed-wall)))
       (xm (point-x field-point))
       (ym (point-y field-point))
       (zm (point-z field-point)))
  (cond ((< (wall-length exposed-wall) step)
         field-point)                            ;No new field points.
        ((not (= x4 x3))
         ;;Don't want to divide by zero in slope calculation.
         (let* ((slope (slope-of-wall exposed-wall))
                (b (- y3 (* slope x3)))
                ;; We will not change zm but only xm and ym.
                (xpc (+ xm (* (cos β2) (* 1 step))))
                (ypc (+ (* slope xpc) b)))
          ;; If we can take a step in the positive direction
          ;; and stay on the wall, we do so.
          (cond ((and (or (> xpc (min x3 x4))(= xpc (min x3 x4)))
                      (or (< xpc (max x3 x4))(= xpc (max x3 x4))))
                 (make-point :x xpc :y ypc :z zm))
                (t field-point))))
         (t (let ((xpc x4)                            ;For vertical lines, all x's are the same.
                  (ypc (+ ym (* 1 step))))
              ;; Try to walk in the positive y direction on the wall
              (cond ((and (or (> ypc (min y3 y4))(= ypc (min y3 y4)))
                          (or (< ypc (max y3 y4))(= ypc (max y3 y4))))
                     (make-point :x xpc :y ypc :z zm))
                    (t field-point))))))
```

A negative step is just a positive step with a negative step size.

```
(defun make-new-field-point-negative-step
       (exposed-wall field-point step β2)
 (make-new-field-point-positive-step
  exposed-wall field-point (* -1 step) β2))
```

The following two functions are used above to search either in the positive or negative directions for the maximum.

```
(defun search-for-max-plus
       (exposed-wall field-point step-size β2 cfmo opening)
 (let* ((new-field-point+ (make-new-field-point-positive-step
                           exposed-wall field-point step-size β2))
        (cf+ (configuration-factor-for-one-opening
              new-field-point+ opening exposed-wall)))
   (cond ((> cf+ (+ cfmo .00001))
          (search-for-max-plus
           exposed-wall new-field-point+
           step-size β2 cf+ opening))
         (t (alter-point field-point         ;We add info to the field point.
                         cf cfmo)
            field-point))))

(defun search-for-max-minus
       (exposed-wall field-point step-size β2 cfmo opening)
 (let* ((new-field-point- (make-new-field-point-positive-step
                           exposed-wall field-point step-size β2))
        (cf- (configuration-factor-for-one-opening
              new-field-point- opening exposed-wall)))
  (cond ((> cf- (+ cfmo .00001))
         (search-for-max-minus
          exposed-wall new-field-point-
          step-size β2 cf- opening))
        (t (alter-point field-point          ;We add info to the field point.
                        cf cfmo)
           field-point))))
```

5. Determine Area to Scan for Maximum

We now wish to determine the area to scan to determine the maximum configuration factor in some region of an exposed wall. We note that the radiation pattern due to a single opening is smooth and has a single peak. The radiation from more than one opening adds incoherently, i.e., their intensities add. Therefore, a little reflection leads one to conclude that the maximum due to openings will lie inside the area determined by wrapping a string around pegs placed at the locations of the individual opeinings' maximums. However, rather than using the area determined with the pegs and string we will determine a rectangular region that just caputures all the maximums. In some cases, such as for openings, there will be smaller rectangular regions excuded from the region to be scanned.

We begin by defining a function to collect the location of all the maximums of the individual openings.The following function takes as input the `list-of-exposing-openings` and the `exposed-wall` and returns a list of the maximum point for the openings. The slot values for these points includes their location and
configuration factor.

```
(defun maximum-of-an-opening-for-n-openings
       (list-of-exposing-openings exposed-wall)
  (map 'list #'(lambda (opening)
                 (Maximum-of-an-Opening opening exposed-wall))
      list-of-exposing-openings))
```

We will have use for the function `sub-list` which take as input an `element` and a `list-of-list`. It returns a list of all the sub-lists that has `element` as a member.

```
(defun sub-list (element list-of-list)
  (remove element list-of-list :test-not #'member))
```

We need two functions. The first determines the separations between the projections into the xy plane of `a-point` and each point in a `list-points`.

```
(defun list-xy-separations (a-point list-points)
  (loop for point in list-points
        collect
          (list point (xy-separation a-point point))))
```

The second function, `min-separation-xy`, uses `list-xy-separations` to determine which point in `list-points` which has the smallest `xy-separation` from `a-point`.

```
(defun min-separation-xy (a-point list-points)
  ;;returns the point nearest a-point
  (let* ((list-point-sep (list-xy-separations a-point list-points))
         (list-sep (map 'list #'second list-point-sep))
         (min-sep (apply 'min list-sep)))
   (loop for point-sep in list-point-sep
         when (= min-sep (second point-sep))
          do
           (return (first point-sep)))))
```

For a region which has no openings in it we proceed as follows. From the list produced by `maximum-of-an-opening-for-n-openings`, we select out the smallest and the largest values for z. Then we use xy-separation to determine the points nearest the projection of the lower left and upper right corners of the exposed region. This then defines a rectangular area to scan that will include the maximum of the openings. We used the rectangular area rather than the area determined by wrapping a string around the pegs because it was easier to determine.

The rectangular regions with excluded region or the holey-rectangular region consists of a rectangular region with rectangular holes cut in it. If a maximum falls into an opening, we need to find a replacement maximum for this errant maximum. This replacement maximum will lie on the boundary of the the opening which the original maximum fell into. Thefore, the region we scan is the rectangular region as determined for a nonholey region with the scanning area expanded to include the opening.

To determine if a maximum falls in an opening, we need a function that determines whether a point lies in the interior of a rectangle. The function `point-in-rectangle` returns the `vertical-rectangle` if the point lies in the rectangle and `nil` if not.

```
(defun point-in-rectangle (vertical-rectangle point)
 (let* ((ll (vertical-rectangle-lower-left vertical-rectangle))
        (ur (vertical-rectangle-upper-right vertical-rectangle))
        (x0 (point-x ll))
        (x1 (point-x ur))
        (z0 (point-z ll))
        (z1 (point-z ur))
        (y0 (point-y ll))
        (y1 (point-y ur))
        (xu (max x0 x1))
        (xll (min x0 x1))
        (yu (max y0 y1))
        (yll (min y0 y1))
        (zu (max z0 z1))
        (zll (min z0 z1))
        (xp (point-x point))
        (yp (point-y point))
        (zp (point-z point)))
  (cond ((not ( = x1 x0))
         (let* ((slope (slope-of-vertical-rectangle vertical-rectangle))
                (b (- y1 (* slope x1))))
          ;; If the point lies on the boundary, it is counted as
          ;; outside the opening
          (cond ((and (and (> zp zll) (< zp zu))
                      (and (> xp xll) (< xp xu))
                      (= (round-off yp 4)
                         (round-off (+ (* slope xp) b) 4)))
                 vertical-rectangle)
                (t nil))))
        (t (cond ((and (and (> zp zll) (< zp zu))
                       (and (> yp yll) (< yp yu))) vertical-rectangle)
                 (t nil))))))
```

Having the results for one opening, we wish to determine if a point lies in any of a list of openings.

```
(defun point-in-an-opening (list-of-openings point)
 (loop for opening in list-of-openings
       do                                    ;A point can only be in only one opening.
  (cond ((point-in-rectangle opening point)
         (return (point-in-rectangle opening point))))
       finally (return nil)))
```

Now suppose we have a list of points and a list of openings and we want a list of openings that have at least one of the points in it.

```
(defun pointed-openings (list-of-openings list-of-points)
 (remove nil (loop for point in list-of-points
                   collect
                    (point-in-an-opening list-of-openings point))))
```

If we have a list of rectangles, we want to generate a list of the lower left and upper right corners.

```
(defun list-of-corners (list-of-rectangles)
  (append
    (map 'list #' vertical-rectangle-lower-left list-of-rectangles)
    (map 'list #' vertical-rectangle-upper-right list-of-rectangles)))
```

We are now in a position to determine the area to be scanned on a wall which has openings in it.

```
(defun area-to-scan-on-wall
      (list-of-exposing-openings exposed-wall list-of-exposed-openings)
  (let* ((list-of-max (maximum-of-an-opening-for-n-openings
                        list-of-exposing-openings exposed-wall))
         (list-of-corners (pointed-openings list-of-exposed-openings
                                              list-of-max))
         ;;If list-of-corners is nil, append does not add it to the list.
         (list-of-max-plus (append list-of-max list-of-corners))
         (list-z (map 'list #'point-z list-of-max-plus))
         (z1 (apply 'min list-z))
         (z2 (apply 'max list-z))
         (ll (wall-lower-left exposed-wall))
         (ur (wall-upper-right exposed-wall))
         (point-near-ll (min-separation-xy ll list-of-max-plus))
         (point-near-ur (min-separation-xy ur list-of-max-plus))
         (n-lower-left (make-point :x (point-x point-near-ll)
                                   :y (point-y point-near-ll) :z z1))
         (n-upper-right (make-point :x (point-x point-near-ur)
                                    :y (point-y point-near-ur) :z z2)))
    (make-vertical-rectangle
      :lower-left n-lower-left
      :upper-right n-upper-right)))
```

6. Maximum Configuration Factor for each subregion of the exposed wall.

We wish to determine the maximum radiation field (or configuration factor) for each subregion of the exposed wall. The size of fields we are interested in are those fields that can produce piloted ignition. If the radiation field is small relative to that required for piloted ignition, we are not interested in it. We can define small by looking for the most vulnerable material that might reasonably be exposed. The most vulnerable combustible target is normally assumed to be oven dried wood [5]. If the maximum possible field is below the piloted ignition level for this material then it is assumed that no exposure problem exists.

To take advantage of this assumption, we define a global constant `*minimum-pilot-ignition-level*` to have this value so we can use it as the default value for the piloted ignition level for materials.

```
(defvar *minimum-pilot-ignition-level* 1.255
 "Minimum piloted ignition level for oven dried wood in W./cm. sq.")
```

To determine the maximum configuration factor we can scan the entire subregion of interest and look for the maximum. However, calculating the configuration factor for all these points of the scan takes a lot of computer computation time. Therefore, it is desirable to hold the number of points we scan to some reasonable minimum. This is achieved by scanning over the area in steps as large as possible, but small enough so we don't make a serious error in determining the maximum.

```
(defun min-step-size (exposed-wall list-of-openings)
 (loop for opening in list-of-openings
       minimize
         (step-size opening exposed-wall)))


(defun number-steps (m-step-size region-to-scan)
  (cond ((or (= m-step-size 0)     ;Don't want to divide by 0.
             (< (/ (vertical-rectangle-width region-to-scan)
                   m-step-size) 1)) 1) ;Want at least one step.
        (t (round (/ (vertical-rectangle-width region-to-scan)
                     m-step-size)))))
```

The maximum configuration factor due to a number of openings is always less than the sum of the maximums of the individual openings. We will have use for this sum so the following function computes it.

```
(defun sum-of-individual-cf (list-of-openings exposed-wall)
 (apply '+ (map 'list #'(lambda (max-point)
                          (point-cf max-point))
                (maximum-of-an-opening-for-n-openings     ;list of max points
                  list-of-openings exposed-wall))))
```

We will scan the wall by using the vertical scan to handle the horizontal scan. If a point falls in an opening, it will be excluded.

```
(defun find-maximum-configuration-factor-opening
      (list-of-exposing-openings exposed-wall
                                              &key list-of-exposed-openings
                                              (minimum-pilot-ignition-level-for-wall
                                               *minimum-pilot-ignition-level*)
                                              ;;the default value
                                              (source-flux 20)
                                              print-scan)
 (let* ((temp-max-point (make-point :x 0 :y 0 :z 0 :cf 0))
        (temp-list-max-point (list temp-max-point)))
  (cond ((null list-of-exposing-openings)
          ;;The location of the point is not important since cf = 0 which
          ;;is the critical information.
          temp-list-max-point)
        ((= 1 (length list-of-exposing-openings))
         ;;This is the simple case of one opening. Then the max cf is
         ;;just what it is for an opening & we are done.
         (list (Maximum-of-an-Opening
                 (first list-of-exposing-openings) exposed-wall)))
        ;;If we have more than one opening we have to do some scanning if
        ;;there is a possibility of exceeding threshold.
        (t (let ((cfm 0)
                 (sum-of-cfs
                  (sum-of-individual-cf list-of-exposing-openings
                                        exposed-wall)))
             (cond ((< (* 1.25 sum-of-cfs)          ;If the sum of all maxs is
                        ;;significantly less than threshold, we don't need to
                        ;;calculate cf but we use the sum.
                        (/ minimum-pilot-ignition-level-for-wall
                           source-flux))
                    ;;We take the minimum-pilot-ignition-level-for-wall and divide it
                    ;;by source-flux, which has a default value of 20 W/cm sq., to get
                    ;;minimum configuration factor.
                    (print 'sum-of-cfs)
                    (setq cfm (round-off sum-of-cfs 4))
                    ;;The following is not the location of the max or its value.
                    ;;It doesn't matter since cf is below threshold.
                    (list (alter-point temp-max-point cf cfm)))
                   ;; We step through the region bounding the individual maxs.
                   ;; The vertical scan includes the horizontal scan.
                   (t (scan-cf-vertically
                        list-of-exposing-openings exposed-wall
                        list-of-exposed-openings print-scan))))))))
```

The following function scans the horizontal scanning function vertically.

```
(defun scan-cf-vertically
      (list-of-exposing-openings exposed-wall
                                              list-of-exposed-openings print-scan)
```

```
 (let* ((region-to-scan
          (area-to-scan-on-wall
           list-of-exposing-openings exposed-wall
           list-of-exposed-openings))
        (z1 (point-z (vertical-rectangle-lower-left region-to-scan)))
        (z2 (point-z (vertical-rectangle-upper-right region-to-scan)))
        (m-step-size (min-step-size exposed-wall
                                    list-of-exposing-openings))
        (xs (point-x (vertical-rectangle-lower-left region-to-scan)))
        (ys (point-y (vertical-rectangle-lower-left region-to-scan)))
        (starting-point
         (make-point :x xs :y ys :z z1))
        (initial-cf (configuration-factor-for-n-openings
                      starting-point list-of-exposing-openings
                      exposed-wall))
        (temp-list-max-point
         (list (make-point :x xs :y ys :z z1
                           :cf initial-cf))))
  ;;For diagnostic purposes it is sometimes of interest to print out
  ;;the area to be scanned.
  (cond (print-scan
          (format t "~%xl = ~2$, yl = ~2$, zl = ~2$" xs ys z1)
          (format t "~%xu = ~2$, yu = ~2$, zu = ~2$"
                  (point-x (vertical-rectangle-upper-right region-to-scan))
                  (point-y (vertical-rectangle-upper-right region-to-scan))
                  z2)))
  ;;Back to business.
  (loop for z from z1 to z2 by m-step-size
        do
   (setq temp-list-max-point
         (scan-cf-horizontally
          z list-of-exposing-openings exposed-wall
          list-of-exposed-openings temp-list-max-point
          m-step-size region-to-scan print-scan))
        finally (return temp-list-max-point))))
```

The following is the horizontal scanning function.

```
(defun scan-cf-horizontally                    ;We fix z and scan over xy.
       (z list-of-exposing-openings exposed-wall
          list-of-exposed-openings temp-list-max-point
          m-step-size region-to-scan print-scan)
 (let* ((x (point-x (vertical-rectangle-lower-left region-to-scan)))
        (y (point-y (vertical-rectangle-lower-left region-to-scan)))
        (field-point (make-point :x x :y y :z z))
        (β2 (slope-of-wall exposed-wall))
        (number-steps (number-steps m-step-size region-to-scan)))
  (loop for k from 0 to number-steps
        do
   (alter-point field-point
                x (+ x (* k m-step-size (cos β2)))
                y (+ y (* k m-step-size (sin β2))))
   ;;If field-point is in an opening, we skip it.
   (cond ((point-in-an-opening list-of-exposed-openings field-point))
         ;;Otherwise we process it.
         (t (let
              ((new-cf
                 (round-off (configuration-factor-for-n-openings
                              field-point list-of-exposing-openings
                              exposed-wall) 4))
               (current-max-cf (point-cf (first
                                           temp-list-max-point))))
                ;;If we make print-scan true, we get a printout of the scan.
                (cond (print-scan
                        (format
                          t
                          "~%x = ~2,0,6$, y = ~2,0,6$, z = ~2,0,6$, new-cf = ~4,0,6$, m-
step-size = ~2,0,6$"
                                (point-x field-point) (point-y field-point) (point-z fie
ld-point)
                                new-cf m-step-size)))
                (cond ((> new-cf current-max-cf) ;replace entire list
                       (setq temp-list-max-point
                             (list (make-point :x (point-x field-point)
                                               :y (point-y field-point)
                                               :z (point-z field-point)
                                               :cf new-cf))))
                      ((= new-cf current-max-cf)  ;add to list
                       (setq temp-list-max-point
                             (cons (make-point :x (point-x field-point)
                                               :y (point-y field-point)
                                               :z (point-z field-point)
                                               :cf new-cf)
                                   temp-list-max-point))))))
         finally (return temp-list-max-point))))
```

To convert to field values from configuration factors we multiply configuration factors by the source's flux. This is such a simple thing we do not need a function to compute it.

7. Conclusions and Possible Future Work

This report documents a program module in the expert system EXPOSURE. It is used for calculating the maximum electromagnetic radiation incident upon a building's wall due to the burning of a neighboring building. It is assumed that the burning building has an arbitrary number of rectangular openings emitting radiation. The exposed wall can be considered as being composed of a number of rectangular regions. These regions may be openings or regions of different material. The results of calculating the configuration factor using the function `configuration-factor-for-one-opening` with analytical expressions for three special cases showed agreement to 4 or 5 decimal places (see Appendix A). This is just little more accuracy that the expert system requires so that is desirable.

This module also allows the consideration of nonparallel exposing and exposed walls and the explicit use of the appropriate wall failure mechanism. In the case of combustible walls, this means using the piloted ignition threshold for the combustible material rather than a very conservative value for all combustible materials such as is done by NFPA 80A. Also, the effect of the location of opening is taken into consideration. Again this is beyond current practice. These features improve the accuracy of the analysis of the safety of buildings as it relates to the spread of fire between buildings and facilitates innovative design and use of materials.

Future work could include work on the selection of the step size used in scanning. In many real world problems an exposing building with many windows exposes a long wall. The present program could probably be made faster by a more intelligent selection of the initial step size and increasing the step size in regions of small configuration factors. Also, much more effort needs to be spend on testing the program before it could be considered being used in a deployed expert system.

References

1. Routley, Gordon "Fire Department Operations" Cote, Arthur E. and Jim L. Linville, *Fire Protection Handbook*, 16 Ed. National Fire Protection Association, Mass. 1986 pp 15-23 to 15-36.

2. NFPA 80A: Recommended Practice for Protection of Buildings from Exterior Fire Exposures. [1980]

3. Protection Against Fire Exposure, Loss Prevention Data 1-20, March 1979, Factory Mutual System

4. McGuire, J. H. "Fire and the Spatial Separation of Buildings." Fire Technology. 1(4): 278-2871965.

5. Law, Margaret, "Heat Radiation from Fires and Building Separation" Joint Fire Research Organization Technical Paper no. 5, 1963

6. Willians-Leir, G. "Approximations for Spatial Separation" Fire Technology, vol 2 no.2 136-1451966

7. Drysdale, Dougal. *An Introduction to Fire Dynamics*. New York: John Wiley and Sons1985. 424 p.

8. Siegel, Robert and John R. Howell, *Thermal Radiation Heat Transfer*. 2nd Ed. McGraw-Hill Book Co. 1981

9. Tein, C. L., K. Y. Lee, and A. J. Stretton "Radiation Heat Transfer" *SFPE Handbook of Fire Protection Engineering*, 1st Ed. pp 1-93 to 1-106 National Fire Protection Association, Mass 1988

Appendix A: Configuration Factor Calculation Compared to Results of Using an Analytic Expression

In this appendix we will compare the program's results for the configuration factor of a single opening with results obtained by the use of analytical expressions for three special cases.

Case 1.

The first special case is when the infinitesimal element is parallel to the opening and located on a line passing through a corner of the opening and the line is perpendicular to the plane of the opening.

The analytical expression which we obtained from reference 8 page 823 #4, is expressed in Lisp code in the function `cf-parallel-corner`.

```
(defun cf-parallel-corner (opening-width opening-height separation)
 (let* ((x (/ opening-width separation))
        (y (/ opening-height separation))
        (sx (sqrt (+ 1 (expt x 2))))
        (sy (sqrt (+ 1 (expt y 2))))
        (tany (atan (/ y sx)))
        (tanx (atan (/ x sy))))
  (/ (+ (/ (* x tany) sx) (/ (* y tanx) sy)) (* 2 *single-p-pi*))))
```

We write the following function to compare the program's result with that obtained by using `cf-parallel-corner`.

```
(defun program-cf-parallel-corner
       (opening-width opening-height separation)
 (let* ((field-point (make-point :x 0 :y separation :z 0))
        ;; Field point is on y axis.
        (llo (make-point :x opening-width :y 0 :z 0))
        (uro (make-point :x 0 :y 0 :z opening-height))
        (opening (make-opening :lower-left llo :upper-right uro))
        (llw (make-point :x -1 :y separation :z -1))
        (urw (make-point :x 1 :y separation :z 1))
        (exposed-wall (make-wall :lower-left llw :upper-right urw)))
  (configuration-factor-for-one-opening
   field-point opening exposed-wall)))

(defun single-comparison-cf-parallel-corner
       (opening-width opening-height separation)
  (let ((analytical
          (round-off
            (cf-parallel-corner opening-width opening-height separation) 6))
        (programs
          (program-cf-parallel-corner
            opening-width opening-height separation)))
    (format t "~%~2$  ~6$   ~6$   ~6$      ~0$" separation analytical programs
            (/ analytical programs) (* (expt 10 6) (- analytical programs)))))


(defun list-comparison-cf-parallel-corner (list-of-separations)
  (format t "~%Y    analytical  program   analyt./prog.  difference x10^6")
 (loop for y in list-of-separations
       do
   (single-comparison-cf-parallel-corner 1 1 y)))
```

If we run the function

```
(list-comparison-cf-parallel-corner '(.25 .5 1 2 3 4 5 6 7))
```

with the indicated arguments, we obtain:

```
Y     analytical   program    analyt./prog.   difference x10^6
0.25   0.237856    0.237857    0.999995        -1.
0.50   0.207757    0.207767    0.999954        -10.
1.00   0.138532    0.138572    0.999710        -40.
2.00   0.059864    0.059909    0.999255        -45.
3.00   0.030829    0.030864    0.998858        -35.
4.00   0.018369    0.018390    0.998866        -21.
5.00   0.012089    0.012105    0.998641        -16.
6.00   0.008527    0.008535    0.999078        -8.
7.00   0.006324    0.006329    0.999242        -5.
```

Case 2.

Configuration factor perpendicular over one corner

The second special case is when the infinitesimal element is perpendicular to the opening and located on a line passing through a corner of the opening and the line is perpendicular to the plane of the opening.

The analytical expression which we obtained from reference 8 page 823 #6, is expressed in Lisp code in the function **cf-perpendicular-corner**.

```
(defun cf-perpendicular-corner
       (opening-width opening-height separation)
 (let* ((x (/ opening-width opening-height))
        (y (/ separation opening-height))
        (sxy (sqrt (+ (expt x 2) (expt y 2))))
        (tany (atan (/ 1 y)))
        (tanxy (atan (/ 1 sxy))))
  (/ (- tany (/ (* y tanxy) sxy))(* 2 *single-p-pi*))))

(defun program-cf-perpendicular-corner
       (opening-width opening-height separation)
 (let* ((field-point (make-point :x 0 :y separation :z 0))
        ;; Field point is on y axis.
        (llo (make-point :x opening-width :y 0 :z 0))
        (uro (make-point :x 0 :y 0 :z opening-height))
        (opening (make-opening :lower-left llo :upper-right uro))
        (llw (make-point :x 0 :y separation :z -1))
        (urw (make-point :x 0 :y (+ separation 1) :z 1))
        (exposed-wall (make-wall :lower-left llw :upper-right urw)))
  (configuration-factor-for-one-opening
   field-point opening exposed-wall)))

(defun single-comparison-cf-perpendicular-corner
       (opening-width opening-height separation)
 (let ((analytical
         (round-off
          (cf-perpendicular-corner opening-width
                                           opening-height separation) 6))
       (programs
         (program-cf-perpendicular-corner
          opening-width opening-height separation)))
   (format t "~%~2$  ~6$   ~6$   ~6$      ~0$" separation analytical programs
           (/ analytical programs) (* (expt 10 6) (- analytical programs)))))


(defun list-comparison-cf-perpendicular-corner
       (list-of-separations)
  (format t "~%Y    analytical  program   analyt./prog.  difference x10^6")
 (loop for y in list-of-separations
       do
  (single-comparison-cf-perpendicular-corner 1 1 y)))
```

If we run the function

```
(list-comparison-cf-perpendicular-corner '(.5 1 2 3))
```

with the indicated arguments, we obtain:

```
Y    analytical  program   analyt./prog.  difference x10^6
0.50  0.124269   0.124326   0.999541      -57.
1.00  0.055734   0.055789   0.999010      -55.
2.00  0.013928   0.013952   0.998309      -24.
3.00  0.004964   0.004976   0.997573      -12.
```

Case 3.

Configuration factor angle theta over one corner

The third special case is when the infinitesimal element is in a plane that has an angle of intersection with the plane of the opening of theta. The separation is the distance from the corner of the opening where the two planes intersect to the infinitesimal element.

Reference "Fire-Safe Structural Steel A Design Guide" American Iron and Steel Institute, (1979) page 24

```
(defun cf-theta-corner (opening-width opening-height separation theta)
 (let* ((x (/ opening-width separation))
        (y (/ opening-height separation))
        (yc (* y (cos theta)))
        (sy (sqrt (- (+ 1 (expt y 2)) (* 2 yc))))
        (xs (sqrt (+ (expt x 2) (expt (sin theta) 2)))))
  (/ (+ (atan x) (/ (* (- yc 1) (atan (/ x sy))) sy)
        (* (* x (cos theta))(/ (+ (atan (/ (- y (cos theta)) xs))
                                  (atan (/ (cos theta) xs))) xs)))
     (* 2 *single-p-pi*))))


(defun program-cf-theta-corner
       (opening-width opening-height separation theta)
 (let* ((field-point (make-point :x (* separation (cos theta))
                                 :y (* separation (sin theta))
                                 :z 0))
        (llo (make-point :x opening-width :y 0 :z 0))
        (uro (make-point :x 0 :y 0 :z opening-height))
        (opening (make-opening :lower-left llo :upper-right uro))
        (llw (make-point :x (* separation (cos theta))
                         :y (* separation (sin theta))
                         :z -1))
        (urw (make-point :x (* (+ separation 1) (cos theta))
                         :y (* (+ separation 1) (sin theta))
                         :z 1))
        (exposed-wall (make-wall :lower-left llw :upper-right urw)))
  (configuration-factor-for-one-opening
   field-point opening exposed-wall)))

(defun single-comparison-cf-theta-corner
       (opening-width opening-height separation theta)
 (let ((analytical
         (round-off
          (cf-theta-corner opening-width
                              opening-height separation
                              theta) 6))
        (programs
          (program-cf-theta-corner
           opening-width opening-height separation theta)))
    (format t "~%~2$  ~3$   ~6$   ~6$   ~6$      ~0$"
            separation theta analytical programs
            (/ analytical programs) (* (expt 10 6)(- analytical programs)))))

(defun list-comparison-cf-theta-corner
       (list-of-separations list-of-thetas)
  (format t "~%Y      theta  analytical  program   analyt./prog.  difference x10^6")
 (loop for xy in list-of-separations
      do
  (loop for theta in list-of-thetas
       do
   (single-comparison-cf-theta-corner 1 1 xy theta))))
```

If we run the function `(list-comparison-cf-theta-corner '(1 2 3) (list (/ *single-p-pi* 2) (* 3 (/ *single-p-pi* 4)) (* 3.5 (/ *single-p-pi* 4))(* 3.9 (/ *single-p-pi* 4))))` with the indicated arguments, we obtain:

```
Y     theta  analytical  program   analyt./prog.  difference x10^6
1.00  1.571   0.055734   0.055789   0.999010      -55.
1.00  2.356   0.013020   0.013042   0.998303      -22.
1.00  2.749   0.003201   0.003207   0.997986      -6.
1.00  3.063   0.000127   0.000128   0.995048      -1.
2.00  1.571   0.013928   0.013952   0.998309      -24.
2.00  2.356   0.003797   0.003810   0.996505      -13.
2.00  2.749   0.000961   0.000965   0.996236      -4.
2.00  3.063   0.000039   0.000039   1.007274      0.
3.00  1.571   0.004964   0.004976   0.997573      -12.
3.00  2.356   0.001542   0.001552   0.993681      -10.
3.00  2.749   0.000401   0.000403   0.993925      -2.
3.00  3.063   0.000016   0.000016   0.980120      -0.
```

Appendix B: Results for Some Simple Buildings

We present here the results of applying the results of this file to three cases that will test the performance of this program. It is not a comprehensive test, but one that should exercise the program's main features.

We define a function that will take as its first argument a list of the list of lower left and upper right corners of the openings. The next six arguments are the coordinates of the lower left and upper right corners of the exposed area.

```
(defun test-find-maximum-configuration-factor-opening
       (list-of-pairs-of-corners-of-openings
         x3 y3 z3 x4 y4 z4                  ;The corner coordinates of the exposed area.
         &key print-scan)                   ;Scan pattern produced upon request.
 (let*
  ((list-of-exposing-openings
    (map 'list
         #'(lambda (list-of-pair-of-corners)
            (let ((ll (make-point :x (first list-of-pair-of-corners)
                                  :y (second list-of-pair-of-corners)
                                  :z (third list-of-pair-of-corners)))
                  (ur (make-point :x (fourth list-of-pair-of-corners)
                                  :y (fifth list-of-pair-of-corners)
                                  :z (sixth list-of-pair-of-corners))))
             (make-opening :lower-left ll :upper-right ur)))
         list-of-pairs-of-corners-of-openings))
   (llw (make-point :x x3
                    :y y3
                    :z z3))
   (urw (make-point :x x4
                    :y y4
                    :z z4))
   (exposed-wall (make-wall :lower-left llw :upper-right urw)))
  (find-maximum-configuration-factor-opening
   list-of-exposing-openings exposed-wall :print-scan print-scan)))
```

We will test three cases. In all cases we have two exposing windows either in the xz or yz plane. While the windows are the same size, they are set in at different heights. For the first two cases the exposed wall is parallel to the exposing wall. In the last case, it is at a small angle to the exposing wall.

Our first test examines the time need to compute the results.

```
(defun test-cases ()
  (values (time (test-find-maximum-configuration-factor-opening
                  '((0 1 1 0 7 7) (0 9 7 0 15 13)) 20 0 0 20 15 15))
          (terpri)
          ;;Interchanging x and y, we get
          (time (test-find-maximum-configuration-factor-opening
                 '((1 0 1 7 0 7) (9 0 7 15 0 13)) 0 20 0 15 20 15))
          (terpri)
          ;;Setting the exposed wall at a small angle to the exposing wall.
          (time (test-find-maximum-configuration-factor-opening
                 '((1 0 1 7 0 7) (9 0 7 15 0 13)) 0 18 0 15 22 15))))
```

The results of running test-cases follows:

```
         (test-cases)
Evaluation of (TEST-FIND-MAXIMUM-CONFIGURATION-FACTOR-OPENING '# 20 0 0 ...)
took 8.441768 seconds of elapsed time including:
  0.112 seconds processing sequence breaks,
  0.360 seconds in the storage system (including 0.052 seconds waiting for pages):
    0.179 seconds processing 313 page faults including 4 fetches,
    0.171 seconds in creating and destroying pages, and
    0.010 seconds in miscellaneous storage system tasks.
The garbage collector has flipped; so no consing was measured.
Evaluation of (TEST-FIND-MAXIMUM-CONFIGURATION-FACTOR-OPENING '# 0 20 0 ...)
took 6.812716 seconds of elapsed time including:
  0.137 seconds processing sequence breaks,
  0.531 seconds in the storage system (including 0.135 seconds waiting for pages):
    0.312 seconds processing 330 page faults including 21 fetches,
    0.159 seconds in creating and destroying pages, and
    0.060 seconds in miscellaneous storage system tasks.
235 list, 40,365 structure, 12,325 stack words consed in WORKING-STORAGE-AREA.
Evaluation of (TEST-FIND-MAXIMUM-CONFIGURATION-FACTOR-OPENING '# 0 18 0 ...)
took 8.779506 seconds of elapsed time including:
  0.173 seconds processing sequence breaks,
  0.587 seconds in the storage system (including 0.082 seconds waiting for pages):
    0.220 seconds processing 391 page faults including 6 fetches,
    0.231 seconds in creating and destroying pages, and
    0.135 seconds in miscellaneous storage system tasks.
221 list, 44,207 structure, 18,121 stack words consed in WORKING-STORAGE-AREA.
(#S(POINT :X 20.0
          :Y 6.828427
          :Z 6.828427
          :CF 247/5000))
NIL
(#S(POINT :X 6.828427
          :Y 20.0
          :Z 6.828427
          :CF 247/5000))
NIL
(#S(POINT :X 2.5196323
          :Y 18.671902
          :Z 6.607681
          :CF 513/10000))
```

The second shows the area to be scanned and the actual scanned points.

```
(defun test-cases-2 ()
  (values (test-find-maximum-configuration-factor-opening
             '((0 1 1 0 7 7) (0 9 7 0 15 13)) 20 0 0 20 15 15 :print-scan t)
          (terpri)
          ;;Interchanging x and y, we get
          (test-find-maximum-configuration-factor-opening
            '((1 0 1 7 0 7) (9 0 7 15 0 13)) 0 20 0 15 20 15 :print-scan t)
          (terpri)
          ;;Setting the exposed wall at a small angle to the exposing wall.
          (test-find-maximum-configuration-factor-opening
            '((1 0 1 7 0 7) (9 0 7 15 0 13))
           0 18 0 15 22 15 :print-scan t)))
```

The results of running **test-cases-2** follows:

```
  (test-cases-2)
xl = 20.00, yl = 4.00, zl = 4.00
xu = 20.00, yu = 12.00, zu = 10.00
x =  20.00, y =   4.00, z =   4.00, new-cf =  .0460, m-step-size =   2.83
x =  20.00, y =   6.83, z =   4.00, new-cf =  .0479, m-step-size =   2.83
x =  20.00, y =   9.66, z =   4.00, new-cf =  .0471, m-step-size =   2.83
x =  20.00, y =  12.49, z =   4.00, new-cf =  .0439, m-step-size =   2.83
x =  20.00, y =   4.00, z =   6.83, new-cf =  .0470, m-step-size =   2.83
x =  20.00, y =   6.83, z =   6.83, new-cf =  .0494, m-step-size =   2.83
x =  20.00, y =   9.66, z =   6.83, new-cf =  .0491, m-step-size =   2.83
```

```
x =  20.00, y =  12.49, z =   6.83, new-cf =  .0462, m-step-size =   2.83
x =  20.00, y =   4.00, z =   9.66, new-cf =  .0450, m-step-size =   2.83
x =  20.00, y =   6.83, z =   9.66, new-cf =  .0478, m-step-size =   2.83
x =  20.00, y =   9.66, z =   9.66, new-cf =  .0481, m-step-size =   2.83
x =  20.00, y =  12.49, z =   9.66, new-cf =  .0457, m-step-size =   2.83

xl = 4.00, yl = 20.00, zl = 4.00
xu = 12.00, yu = 20.00, zu = 10.00

x =   6.83, y =  20.00, z =   4.00, new-cf =  .0479, m-step-size =   2.83
x =   9.66, y =  20.00, z =   4.00, new-cf =  .0471, m-step-size =   2.83
x =  12.49, y =  20.00, z =   4.00, new-cf =  .0439, m-step-size =   2.83
x =   4.00, y =  20.00, z =   6.83, new-cf =  .0470, m-step-size =   2.83
x =   6.83, y =  20.00, z =   6.83, new-cf =  .0494, m-step-size =   2.83
x =   9.66, y =  20.00, z =   6.83, new-cf =  .0491, m-step-size =   2.83
x =  12.49, y =  20.00, z =   6.83, new-cf =  .0462, m-step-size =   2.83
x =   4.00, y =  20.00, z =   9.66, new-cf =  .0450, m-step-size =   2.83
x =   6.83, y =  20.00, z =   9.66, new-cf =  .0478, m-step-size =   2.83
x =   9.66, y =  20.00, z =   9.66, new-cf =  .0481, m-step-size =   2.83
x =  12.49, y =  20.00, z =   9.66, new-cf =  .0457, m-step-size =   2.83

xl = 0.00, yl = 18.00, zl = 4.00
xu = 6.72, yu = 19.79, zu = 10.00
x =    .00, y =  18.00, z =   4.00, new-cf =  .0485, m-step-size =   2.61
x =   2.52, y =  18.67, z =   4.00, new-cf =  .0503, m-step-size =   2.61
x =   5.04, y =  19.34, z =   4.00, new-cf =  .0495, m-step-size =   2.61
x =   7.56, y =  20.02, z =   4.00, new-cf =  .0463, m-step-size =   2.61
x =    .00, y =  18.00, z =   6.61, new-cf =  .0490, m-step-size =   2.61
x =   2.52, y =  18.67, z =   6.61, new-cf =  .0513, m-step-size =   2.61
x =   5.04, y =  19.34, z =   6.61, new-cf =  .0509, m-step-size =   2.61
x =   7.56, y =  20.02, z =   6.61, new-cf =  .0481, m-step-size =   2.61
x =    .00, y =  18.00, z =   9.22, new-cf =  .0467, m-step-size =   2.61
x =   2.52, y =  18.67, z =   9.22, new-cf =  .0494, m-step-size =   2.61
x =   5.04, y =  19.34, z =   9.22, new-cf =  .0495, m-step-size =   2.61
x =   7.56, y =  20.02, z =   9.22, new-cf =  .0473, m-step-size =   2.61
(#S(POINT :X 20.0
          :Y 6.828427
          :Z 6.828427
          :CF 247/5000))
NIL
(#S(POINT :X 6.828427
          :Y 20.0
          :Z 6.828427
          :CF 247/5000))
NIL
(#S(POINT :X 2.5196323
          :Y 18.671902
          :Z 6.607681
          :CF 513/10000))
```

NIST-114A (REV. 3-90) U.S. DEPARTMENT OF COMMERCE NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY **BIBLIOGRAPHIC DATA SHEET**	1. PUBLICATION OR REPORT NUMBER NISTIR 4437
	2. PERFORMING ORGANIZATION REPORT NUMBER
	3. PUBLICATION DATE November 1990

4. TITLE AND SUBTITLE

A Program for Calculating the Maximum Radiation on a Wall

5. AUTHOR(S)

Richard L. Smith

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS) U.S. DEPARTMENT OF COMMERCE NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY GAITHERSBURG, MD 20899	7. CONTRACT/GRANT NUMBER
	8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

U.S. Air Force Engineering and Services Center
Airbase Fire Protection and Crash Rescue System Branch
Tyndall AFB, FL 32403-6001

10. SUPPLEMENTARY NOTES

11. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

This report describes a program module of the expert system EXPOSURE. This module, written in Common Lisp, is for calculating the maximum electromagnetic radiation incident upon a building's wall due to theburning of a neighboring building. It is assumed that the burning building has an arbitrary number of rectangular openings emitting radiation. The exposed wall can be considered as being composed of a number of rectangular regions. These regions may be openings on regions of different materials. This program can determine the maximum radiation for each of these regions and for the wall less these regions.

12. KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

exposure; radiation; radiative h eat transfer; computer programs

13. AVAILABILITY		14. NUMBER OF PRINTED PAGES 33
X	UNLIMITED	
	FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).	
	ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.	15. PRICE A03
X	ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.	

ELECTRONIC FORM