The GridFire Fire Behavior Model

Gary W. Johnson, Ph.D., David Saah, Ph.D., Max Moritz, Ph.D., Kenneth Cheung

1 Preface

This document is a Literate Program[1], containing both the source code of the software it describes as well as the rationale used in each step of its design and implementation. The purpose of this approach is to enable anyone sufficiently experienced in programming to easily retrace the author's footsteps as they read through the text and code. By the time they have reached the end of this document, the reader should have just as strong a grasp of the system as the original programmer.

To execute the code illustrated within this document, you will need to install several pieces of software, all of which are open source and/or freely available for all major operating systems. These programs are listed in Table 1 along with their minimum required versions and URLs from which they may be downloaded.

Table 1: Software necessary to evaluate the code in this document

Name	Version	URL
Java Development Kit	11+	`https://jdk.java.net`
Clojure CLI Tools	1.10+	`https://clojure.org/guides/getting_started`
Postgresql	10+	`https://www.postgresql.org/download`
PostGIS	3+	`https://postgis.net/install`

GridFire is written in the Clojure programming language[2], which is a modern dialect of Lisp hosted on the Java Virtual Machine.(Hickey, 2008) As a result, a Java Development Kit is required to compile and run the code shown throughout this document.

The Clojure CLI tools are used to download required libraries and provide a code evaluation prompt (a.k.a. REPL) into which we will enter the code making up this fire model.

Postgresql (along with the PostGIS spatial extensions) will be used to load and serve raster-formatted GIS layers to the GridFire program. Although it is beyond the scope of this document, PostGIS[3] provides a rich API for manipulating both raster and vector layers through SQL.

[1] `https://en.wikipedia.org/wiki/Literate_programming`
[2] `https://clojure.org`
[3] `https://postgis.net`
[4] `https://www.eclipse.org/legal/epl-2.0/`

2 Setting Up the Clojure Environment

Because Clojure is implemented on the Java Virtual Machine (JVM), we must explicitly list all of the libraries used by our program on the Java classpath. Fortunately, the Clojure CLI tools can handle downloading and storing these libraries as well as making them available to the Clojure process at runtime. However, in order for Clojure to know which libraries are needed, we must first create its build configuration file, called “deps.edn”, and place it in the directory from which we will call our Clojure program. The complete deps.edn for the current GridFire version is shown below.

```
{:paths ["src" "resources"]

 :deps {com.nextjournal/beholder           {:mvn/version "1.0.0"}
        com.taoensso/tufte                 {:mvn/version "2.2.0"}
        kixi/stats                         {:mvn/version "0.5.4"}
        net.mikera/core.matrix             {:mvn/version "0.62.0"}
        net.mikera/vectorz-clj             {:mvn/version "0.48.0"}
        org.clojars.lambdatronic/matrix-viz {:mvn/version "0.1.7"}
        org.clojure/clojure                {:mvn/version "1.10.3"}
        org.clojure/core.async             {:mvn/version "1.3.622"}
        org.clojure/data.csv               {:mvn/version "1.0.0"}
        org.clojure/data.json              {:mvn/version "2.4.0"}
        org.clojure/java.jdbc              {:mvn/version "0.7.12"}
        org.clojure/tools.cli              {:mvn/version "1.0.206"}
        org.postgresql/postgresql          {:mvn/version "42.2.23"}
        sig-gis/magellan                   {:mvn/version "20210401"}
        sig-gis/triangulum                 {:git/url "https://github.com/sig-gis/triangulum"
                                            :sha     "5b179a97ebd8fbcbff51776db06d9770cb649b9d"}}

 :mvn/repos {"osgeo" {:url "https://repo.osgeo.org/repository/release/"}}

 :aliases {:build-test-db     {:extra-paths ["test"]
                               :main-opts   ["-m" "gridfire.build-test-db"]}
           :run               {:main-opts ["-m" "gridfire.cli"]}
           :repl              {:main-opts ["-e" "(require,'gridfire.core)"
                                           "-e" "(in-ns,'gridfire.core)"
                                           "-r"]}
           :make-uberjar      {:replace-deps {com.github.seancorfield/depstar {:mvn/version "2.1.303"}}
                               :exec-fn      hf.depstar/uberjar
                               :exec-args    {:jar         target/gridfire-2021.11.02.jar
                                              :main-class  gridfire.cli
                                              :aot         true
                                              :sync-pom    true
                                              :group-id    sig-gis
                                              :artifact-id gridfire
                                              :version     "2021.11.02"
                                              :manifest
                                              {:specification-title    "Java Advanced Imaging Image I/O Tools"
                                               :specification-version  "1.1"
                                               :specification-vendor   "Sun Microsystems, Inc."
                                               :implementation-title   "com.sun.media.imageio"
                                               :implementation-version "1.1"
                                               :implementation-vendor  "Sun Microsystems, Inc."}}}
           :test              {:extra-paths ["test"]
                               :extra-deps  {com.cognitect/test-runner
                                             {:git/url "https://github.com/cognitect-labs/test-runner.git"
                                              :sha     "dd6da11611eeb87f08780a30ac8ea6012d4c05ce"}}
                               :main-opts   ["-e" "(do,(set!,*warn-on-reflection*,true),nil)"
                                             "-m" "cognitect.test-runner"]}
           :check-reflections {:extra-paths ["test"]
                               :main-opts   ["-e" "(do,(set!,*warn-on-reflection*,true),nil)"
                                             "-e" "(require,'gridfire.cli)"
                                             "-e" "(require,'gridfire.build-test-db)"]}
           :check-deps        {:extra-deps {olical/depot {:mvn/version "2.3.0"}}
                               :main-opts  ["-m" "depot.outdated.main"]}}}
```

Once this file is created, we need to instruct Clojure to download these library dependencies and then run the built-in test suite to verify that GridFire compiles and runs as expected on our local computer.

Before we run the tests, we'll need to set up a test database and import some rasters into it. We will be prompted for the postgres and gridfire_test users' passwords. The postgres user's password will be whatever it is when we set up Postgresql. For the gridfire_test user's password, refer to "src/sql/create_test_db.sql". The default value is simply "gridfire_test".

The following command builds the test database:

```
clojure -M:build-test-db
```

Once that has completed, you can run the following command to launch the test suite:

```
clojure -M:test
```

3 Setting Up the PostGIS Database

GridFire may make use of any raster-formatted GIS layers that are loaded into a PostGIS database. Therefore, we must begin by creating a spatially-enabled database on our local Postgresql server.

When installing Postgresql, we should have been prompted to create an initial superuser called **postgres**, who has full permissions to create new databases and roles. We can log into the Postgresql server as this user with the following **psql** command.

```
psql -U postgres
```

Once logged in, we issue the following commands to first create a new database role and to then create a new database (owned by this role) in which to store our raster data. Finally, we import the PostGIS spatial extensions into the new database.

```
CREATE ROLE gridfire WITH LOGIN CREATEDB;
CREATE DATABASE gridfire WITH OWNER gridfire;
\c gridfire
CREATE EXTENSION postgis;
```

4 Importing Rasters into the Database

Whenever we want to add a new raster-formatted GIS layer to our database, we can simply issue the **raster2pgsql** command as follows, replacing the raster name and table name to match our own datasets.

```
SRID=4326
RASTER=dem.tif
TABLE=dem
DATABASE=gridfire
raster2pgsql -s $SRID $RASTER $TABLE | psql $DATABASE
```

Note: The raster2pgsql command has several useful command line options, including automatic tiling of the raster layer in the database, creating fast spatial indeces after import, or setting raster constraints on the newly created table. Run **raster2pgsql -?** from the command line for more details.

Here's an example shell script that will tile multiple large rasters (asp.tif, cbd.tif, cbh.tif, etc) into 100x100 tiles and import them into our database.

Note: Here we specified a schema (e.g, landfire) along with the table name so as to match the sample config file in “resources/sample_config.edn”.

First create the schema in our database.

```
CREATE SCHEMA landfire;
```

Then we can use the following script to import LANDFIRE layers into our database given the username and schema as inputs.

Note: This script needs to be run in the same folder as where these rasters reside. The filenames of these rasters should match the elements in the for loop (i.e. asp.tif, cbd.tif etc)

```
#!/usr/bin/env bash

USERNAME=$1
SCHEMA=$2
SRID=$3

for LAYER in asp cbd cbh cc ch dem fbfm13 fbfm40 slp
do
    raster2pgsql -t auto -I -C -s $SRID $LAYER.tif $SCHEMA.$LAYER | psql -h localhost -U $USERNAME
done
```

To run the script, give it our username, schema, and srid we wish the layers to have.

```
sh import_landfire_rasters.sh gridfire landfire 90914
```

Whenever we want to add a new spatial reference system to our database, we can insert a record into our spatial_ref_sys table.

```
INSERT INTO public.spatial_ref_sys (srid, auth_name, auth_srid, srtext, proj4text)
VALUES (900914, 'user-generated', 900914,
        'PROJCS["USA_Contiguous_Albers_Equal_Area_Conic_USGS_version",' ||
        'GEOGCS["NAD83",' ||
        'DATUM["North_American_Datum_1983",' ||
        'SPHEROID["GRS 1980",6378137,298.2572221010002,' ||
        'AUTHORITY["EPSG","7019"]],' ||
        'AUTHORITY["EPSG","6269"]],' ||
        'PRIMEM["Greenwich",0],' ||
        'UNIT["degree",0.0174532925199433],' ||
        'AUTHORITY["EPSG","4269"]],' ||
        'PROJECTION["Albers_Conic_Equal_Area"],' ||
        'PARAMETER["standard_parallel_1",29.5],' ||
        'PARAMETER["standard_parallel_2",45.5],' ||
        'PARAMETER["latitude_of_center",23],' ||
        'PARAMETER["longitude_of_center",-96],' ||
        'PARAMETER["false_easting",0],' ||
        'PARAMETER["false_northing",0],' ||
        'UNIT["metre",1,' ||
        'AUTHORITY["EPSG","9001"]]]',
        '+proj=aea +lat_1=29.5 +lat_2=45.5 +lat_0=23 +lon_0=-96 +x_0=0 +y_0=0' ||
        ' +datum=NAD83 +units=m +no_defs');
```

We may also want to import initial ignition rasters into our database. We can do so with a similar script as importing LANDFIRE rasters.

First create a new schema.

```
CREATE SCHEMA ignition;
```

Then we can use the following script to import an ignition raster into our database given the schema and username as inputs.

Note: This script needs to be run in the same folder as where this raster resides. The filename of this raster should match the value assigned to the LAYER variable (i.e., ign) plus a .tif extension.

```
#!/usr/bin/env bash

USERNAME=$1
SCHEMA=$2
SRID=$3

LAYER="ign"
raster2pgsql -I -C -t auto -s $SRID $LAYER.tif $SCHEMA.$LAYER | psql -h localhost -U $USERNAME
```

To run the script, give it the username, schema name, and srid we wish the layers to have.

```
sh import_ignition_rasters.sh gridfire ignition 90014
```

We may also want to import weather rasters into our database. We can do so with a similar script as importing LANDFIRE rasters.

First create a new schema.

```
CREATE SCHEMA weather;
```

Then we can use the following script to import weather rasters into our database given the schema and username as inputs.

Note: This script needs to be run in the same folder as where this rasters resides. The filename of these rasters should match the elements in the for loop (i.e. tmpf_to_sample.tif)

```
#!/usr/bin/env bash

USERNAME=$1
SCHEMA=$2
SRID=$3
TILING=$4

for LAYER in tmpf wd ws rh
do
    if [ -z "$TILING" ]
    then
        raster2pgsql -I -C -t auto -s $SRID ${LAYER}_to_sample.tif $SCHEMA.$LAYER | psql -h localhost -U $USERNAME
    else
        raster2pgsql -I -C -t $TILING -s $SRID ${LAYER}_to_sample.tif $SCHEMA.$LAYER | psql -h localhost -U $USERNAME

    fi
done
```

To run the script, give it the username, schema name, and srid we wish the layers to have.

```
sh import_weather_rasters.sh gridfire weather 90014
```

You may optionally include a fourth argument to set the tiling (defaults to auto).

```
sh import_weather_rasters.sh gridfire weather 90014 800x800
```

Note: This script needs to be run in the same folder as where these rasters reside.

5 Fire Spread Model

GridFire implements the following fire behavior formulas from the fire science literature:

- Surface Fire Spread: Rothermel 1972 with FIREMODS adjustments from Albini 1976
- Crown Fire Initiation: Van Wagner 1977
- Passive/Active Crown Fire Spread: Cruz 2005
- Flame Length and Fire Line Intensity: Byram 1959
- Midflame Wind Adjustment Factor: Albini & Baughman 1979 parameterized as in BehavePlus, FARSITE, FlamMap, FSPro, and FPA according to Andrews 2012
- Fire Spread on a Raster Grid: Morais 2001 (method of adaptive timesteps and fractional distances)
- Spot Fire: Perryman 2013

The following fuel models are supported:

- Anderson 13: no dynamic loading
- Scott & Burgan 40: dynamic loading implemented according to Scott & Burgan 2005

The method used to translate linear fire spread rates to a 2-dimensional raster grid were originally developed by Marco Morais at UCSB as part of his HFire system.(Peterson et al., 2011, 2009, Morais, 2001) Detailed information about this software, including its source code and research article references can be found here:

http://firecenter.berkeley.edu/hfire/about.html

Outputs from GridFire include fire size (ac), fire line intensity (Btu/ft/s), flame length (ft), fire volume (ac*ft), fire shape (ac/ft) and conditional burn probability (times burned/fires initiated). Fire line intensity and flame length may both be exported as either average values per fire or as maps of the individual values per burned cell.

In the following sections, we describe the operation of this system in detail.

5.1 Fuel Model Definitions

All fires ignite and travel through some form of burnable fuel. Although the effects of wind and slope on the rate of fire spread can be quite pronounced, its fundamental thermodynamic characteristics are largely determined by the fuel type in which it is sustained. For wildfires, these fuels are predominantly herbaceous and woody vegetation (both alive and dead) as well as decomposing elements of dead vegetation, such as duff or leaf litter. To estimate the heat output and rate of spread of a fire burning through any of these fuels, we must determine those physical properties that affect heat absorption and release.

Of course, measuring these fuel properties for every kind of vegetation that may be burned in a wildfire is an intractable task. To cope with this, fuels are classified into categories called “fuel models” which share similar burning characteristics. Each fuel model is then assigned a set of representative values for each of the thermally relevant physical properties shown in Table 2.

Note: While M_f is not, in fact, directly assigned to any of these fuel models, their definitions remain incomplete for the purposes of fire spread modelling (particularly those reliant on the curing formulas of dynamic fuel loading) until it is provided as a characteristic of local weather conditions.

Table 2: Physical properties assigned to each fuel model

Property	Description	Units
δ	fuel depth	ft
w_o	ovendry fuel loading	lb/ft^2
σ	fuel particle surface-area-to-volume ratio	ft^2/ft^3
M_x	moisture content of extinction	lb moisture/lb ovendry wood
h	fuel particle low heat content	Btu/lb
ρ_p	ovendry particle density	lb/ft^3
S_T	fuel particle total mineral content	lb minerals/lb ovendry wood
S_e	fuel particle effective mineral content	lb silica-free minerals/lb ovendry wood
M_f	fuel particle moisture content	lb moisture/lb ovendry wood

The fuel models supported by GridFire include the standard 13 fuel models of Rothermel, Albini, and Anderson(Anderson, 1982) and the additional 40 fuel models defined by Scott and Burgan(Scott and Burgan, 2005). These are all concisely encoded in an internal data structure, which may be updated to include additional custom fuel models desired by the user.

```
(ns gridfire.fuel-models)

(def fuel-models
  "Lookup table including one entry for each of the Anderson 13 and
   Scott & Burgan 40 fuel models. The fields have the following
   meanings:
   {number
    [name delta M_x-dead h
     [w_o-dead-1hr w_o-dead-10hr w_o-dead-100hr w_o-live-herbaceous w_o-live-woody]
     [sigma-dead-1hr sigma-dead-10hr sigma-dead-100hr sigma-live-herbaceous sigma-live-woody]]
   }"
  {
   ;; Grass and Grass-dominated (short-grass,timber-grass-and-understory,tall-grass)
   1   [:R01 1.0 12 8 [0.0340 0.0000 0.0000 0.0000 0.0000] [3500.0   0.0  0.0    0.0    0.0]]
   2   [:R02 1.0 15 8 [0.0920 0.0460 0.0230 0.0230 0.0000] [3000.0 109.0 30.0 1500.0    0.0]]
   3   [:R03 2.5 25 8 [0.1380 0.0000 0.0000 0.0000 0.0000] [1500.0   0.0  0.0    0.0    0.0]]
   ;; Chaparral and Shrubfields (chaparral,brush,dormant-brush-hardwood-slash,southern-rough)
   4   [:R04 6.0 20 8 [0.2300 0.1840 0.0920 0.2300 0.0000] [2000.0 109.0 30.0 1500.0    0.0]]
   5   [:R05 2.0 20 8 [0.0460 0.0230 0.0000 0.0920 0.0000] [2000.0 109.0  0.0 1500.0    0.0]]
   6   [:R06 2.5 25 8 [0.0690 0.1150 0.0920 0.0000 0.0000] [1750.0 109.0 30.0    0.0    0.0]]
   7   [:R07 2.5 40 8 [0.0520 0.0860 0.0690 0.0170 0.0000] [1750.0 109.0 30.0 1550.0    0.0]]
   ;; Timber Litter (closed-timber-litter,hardwood-litter,timber-litter-and-understory)
   8   [:R08 0.2 30 8 [0.0690 0.0460 0.1150 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   9   [:R09 0.2 25 8 [0.1340 0.0190 0.0070 0.0000 0.0000] [2500.0 109.0 30.0    0.0    0.0]]
   10  [:R10 1.0 25 8 [0.1380 0.0920 0.2300 0.0920 0.0000] [2000.0 109.0 30.0 1500.0    0.0]]
   ;; Logging Slash (light-logging-slash,medium-logging-slash,heavy-logging-slash)
   11  [:R11 1.0 15 8 [0.0690 0.2070 0.2530 0.0000 0.0000] [1500.0 109.0 30.0    0.0    0.0]]
   12  [:R12 2.3 20 8 [0.1840 0.6440 0.7590 0.0000 0.0000] [1500.0 109.0 30.0    0.0    0.0]]
   13  [:R13 3.0 25 8 [0.3220 1.0580 1.2880 0.0000 0.0000] [1500.0 109.0 30.0    0.0    0.0]]
   ;; Nonburnable (NB)
   91  [:NB1 0.0  0 0 [0.0000 0.0000 0.0000 0.0000 0.0000] [   0.0   0.0  0.0    0.0    0.0]]
   92  [:NB2 0.0  0 0 [0.0000 0.0000 0.0000 0.0000 0.0000] [   0.0   0.0  0.0    0.0    0.0]]
   93  [:NB3 0.0  0 0 [0.0000 0.0000 0.0000 0.0000 0.0000] [   0.0   0.0  0.0    0.0    0.0]]
   98  [:NB4 0.0  0 0 [0.0000 0.0000 0.0000 0.0000 0.0000] [   0.0   0.0  0.0    0.0    0.0]]
   99  [:NB5 0.0  0 0 [0.0000 0.0000 0.0000 0.0000 0.0000] [   0.0   0.0  0.0    0.0    0.0]]
   ;; Grass (GR)
   101 [:GR1 0.4 15 8 [0.0046 0.0000 0.0000 0.0138 0.0000] [2200.0 109.0 30.0 2000.0    0.0]]
   102 [:GR2 1.0 15 8 [0.0046 0.0000 0.0000 0.0459 0.0000] [2000.0 109.0 30.0 1800.0    0.0]]
   103 [:GR3 2.0 30 8 [0.0046 0.0184 0.0000 0.0689 0.0000] [1500.0 109.0 30.0 1300.0    0.0]]
   104 [:GR4 2.0 15 8 [0.0115 0.0000 0.0000 0.0872 0.0000] [2000.0 109.0 30.0 1800.0    0.0]]
   105 [:GR5 1.5 40 8 [0.0184 0.0000 0.0000 0.1148 0.0000] [1800.0 109.0 30.0 1600.0    0.0]]
   106 [:GR6 1.5 40 9 [0.0046 0.0000 0.0000 0.1561 0.0000] [2200.0 109.0 30.0 2000.0    0.0]]
   107 [:GR7 3.0 15 8 [0.0459 0.0000 0.0000 0.2479 0.0000] [2000.0 109.0 30.0 1800.0    0.0]]
   108 [:GR8 4.0 30 8 [0.0230 0.0459 0.0000 0.3352 0.0000] [1500.0 109.0 30.0 1300.0    0.0]]
   109 [:GR9 5.0 40 8 [0.0459 0.0459 0.0000 0.4132 0.0000] [1800.0 109.0 30.0 1600.0    0.0]]
   ;; Grass-Shrub (GS)
   121 [:GS1 0.9 15 8 [0.0092 0.0000 0.0000 0.0230 0.0298] [2000.0 109.0 30.0 1800.0 1800.0]]
   122 [:GS2 1.5 15 8 [0.0230 0.0230 0.0000 0.0275 0.0459] [2000.0 109.0 30.0 1800.0 1800.0]]
   123 [:GS3 1.8 40 8 [0.0138 0.0115 0.0000 0.0666 0.0574] [1800.0 109.0 30.0 1600.0 1600.0]]
   124 [:GS4 2.1 40 8 [0.0872 0.0138 0.0046 0.1561 0.3260] [1800.0 109.0 30.0 1600.0 1600.0]]
   ;; Shrub (SH)
   141 [:SH1 1.0 15 8 [0.0115 0.0115 0.0000 0.0069 0.0597] [2000.0 109.0 30.0 1800.0 1600.0]]
   142 [:SH2 1.0 15 8 [0.0620 0.1102 0.0344 0.0000 0.1768] [2000.0 109.0 30.0    0.0 1600.0]]
   143 [:SH3 2.4 40 8 [0.0207 0.1377 0.0000 0.0000 0.2847] [1600.0 109.0 30.0    0.0 1400.0]]
   144 [:SH4 3.0 30 8 [0.0390 0.0528 0.0092 0.0000 0.1171] [2000.0 109.0 30.0 1800.0 1600.0]]
   145 [:SH5 6.0 15 8 [0.1653 0.0964 0.0000 0.0000 0.1331] [ 750.0 109.0 30.0    0.0 1600.0]]
   146 [:SH6 2.0 30 8 [0.1331 0.0666 0.0000 0.0000 0.0643] [ 750.0 109.0 30.0    0.0 1600.0]]
   147 [:SH7 6.0 15 8 [0.1607 0.2433 0.1010 0.0000 0.1561] [ 750.0 109.0 30.0    0.0 1600.0]]
   148 [:SH8 3.0 40 8 [0.0941 0.1561 0.0390 0.0000 0.1997] [ 750.0 109.0 30.0    0.0 1600.0]]
   149 [:SH9 4.4 40 8 [0.2066 0.1125 0.0000 0.0712 0.3214] [ 750.0 109.0 30.0 1800.0 1500.0]]
   ;; Timber-Understory (TU)
   161 [:TU1 0.6 20 8 [0.0092 0.0413 0.0689 0.0092 0.0413] [2000.0 109.0 30.0 1800.0 1600.0]]
   162 [:TU2 1.0 30 8 [0.0436 0.0826 0.0574 0.0000 0.0092] [2000.0 109.0 30.0    0.0 1600.0]]
   163 [:TU3 1.3 30 8 [0.0505 0.0069 0.0115 0.0298 0.0505] [1800.0 109.0 30.0 1600.0 1400.0]]
   164 [:TU4 0.5 12 8 [0.2066 0.0000 0.0000 0.0000 0.0918] [2300.0 109.0 30.0    0.0 2000.0]]
   165 [:TU5 1.0 25 8 [0.1837 0.1837 0.1377 0.0000 0.1377] [1500.0 109.0 30.0    0.0  750.0]]
   ;; Timber Litter (TL)
   181 [:TL1 0.2 30 8 [0.0459 0.1010 0.1653 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   182 [:TL2 0.2 25 8 [0.0643 0.1056 0.1010 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   183 [:TL3 0.3 20 8 [0.0230 0.1010 0.1286 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   184 [:TL4 0.4 25 8 [0.0230 0.0689 0.1928 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   185 [:TL5 0.6 25 8 [0.0528 0.1148 0.2020 0.0000 0.0000] [2000.0 109.0 30.0    0.0 1600.0]]
   186 [:TL6 0.3 25 8 [0.1102 0.0551 0.0551 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   187 [:TL7 0.4 25 8 [0.0138 0.0643 0.3719 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   188 [:TL8 0.3 35 8 [0.2663 0.0643 0.0505 0.0000 0.0000] [1800.0 109.0 30.0    0.0    0.0]]
   189 [:TL9 0.6 35 8 [0.3053 0.1515 0.1905 0.0000 0.0000] [1800.0 109.0 30.0    0.0 1600.0]]
   ;; Slash-Blowdown (SB)
   201 [:SB1 1.0 25 8 [0.0689 0.1377 0.5051 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   202 [:SB2 1.0 25 8 [0.2066 0.1951 0.1837 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   203 [:SB3 1.2 25 8 [0.2525 0.1263 0.1377 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   204 [:SB4 2.7 25 8 [0.2410 0.1607 0.2410 0.0000 0.0000] [2000.0 109.0 30.0    0.0    0.0]]
   })
```

Once fuel moisture is added to the base fuel model definitions, they will each contain values for the following six fuel size classes:

1. Dead 1 hour (< 1/4" diameter)
2. Dead 10 hour (1/4"–1" diameter)
3. Dead 100 hour (1"–3" diameter)
4. Dead herbaceous (dynamic fuel models only)
5. Live herbaceous
6. Live woody

In order to more easily encode mathematical operations over these size classes, we define a collection of utility functions that will later be used in both the fuel moisture and fire spread algorithms.

```
(defn map-category [f]
  {:dead (f :dead) :live (f :live)})

(defn map-size-class [f]
  {:dead {:1hr        (f :dead :1hr)
          :10hr       (f :dead :10hr)
          :100hr      (f :dead :100hr)
          :herbaceous (f :dead :herbaceous)}
   :live {:herbaceous (f :live :herbaceous)
          :woody      (f :live :woody)}})

(defn category-sum ^double [f]
  (+ ^double (f :dead) ^double (f :live)))

(defn size-class-sum [f]
  {:dead (+ ^double (f :dead :1hr) ^double (f :dead :10hr) ^double (f :dead :100hr) ^double (f :dead :herbaceous))
   :live (+ ^double (f :live :herbaceous) ^double (f :live :woody))})
```

Using these new size class processing functions, we can translate the encoded fuel model definitions into human-readable representations of the fuel model properties.

```
(defn build-fuel-model
  [fuel-model-number]
  (let [[name delta ^double M_x-dead ^double h
         [w_o-dead-1hr w_o-dead-10hr w_o-dead-100hr
          w_o-live-herbaceous w_o-live-woody]
         [sigma-dead-1hr sigma-dead-10hr sigma-dead-100hr
          sigma-live-herbaceous sigma-live-woody]]
        (fuel-models fuel-model-number)
        M_x-dead (* M_x-dead 0.01)
        h        (* h 1000.0)]
    {:name   name
     :number fuel-model-number
     :delta  delta
     :M_x    {:dead {:1hr        M_x-dead
                     :10hr       M_x-dead
                     :100hr      M_x-dead
                     :herbaceous 0.0}
              :live {:herbaceous 0.0
                     :woody      0.0}}
     :w_o    {:dead {:1hr        w_o-dead-1hr
                     :10hr       w_o-dead-10hr
                     :100hr      w_o-dead-100hr
                     :herbaceous 0.0}
              :live {:herbaceous w_o-live-herbaceous
                     :woody      w_o-live-woody}}
     :sigma  {:dead {:1hr        sigma-dead-1hr
                     :10hr       sigma-dead-10hr
                     :100hr      sigma-dead-100hr
                     :herbaceous 0.0}
              :live {:herbaceous sigma-live-herbaceous
                     :woody      sigma-live-woody}}
     :h      {:dead {:1hr        h
                     :10hr       h
                     :100hr      h
                     :herbaceous h}
              :live {:herbaceous h
                     :woody      h}}
     :rho_p  {:dead {:1hr        32.0
                     :10hr       32.0
                     :100hr      32.0
                     :herbaceous 32.0}
              :live {:herbaceous 32.0
                     :woody      32.0}}
     :S_T    {:dead {:1hr        0.0555
                     :10hr       0.0555
                     :100hr      0.0555
                     :herbaceous 0.0555}
              :live {:herbaceous 0.0555
                     :woody      0.0555}}
     :S_e    {:dead {:1hr        0.01
                     :10hr       0.01
                     :100hr      0.01
                     :herbaceous 0.01}
              :live {:herbaceous 0.01
                     :woody      0.01}}}))
```

Although most fuel model properties are static with respect to environmental conditions, the fuel moisture content can have two significant impacts on a fuel model's burning potential:

1. Dynamic fuel loading
2. Live moisture of extinction

These two topics are discussed in the remainder of this section.

5.1.1 Dynamic Fuel Loading

All of the Scott & Burgan 40 fuel models with a live herbaceous component are considered dynamic. In these models, a fraction of the live herbaceous load is transferred to a new dead herbaceous category as a function of live herbaceous moisture content (see equation below).(Burgan, 1979) The dead herbaceous category uses the dead 1 hour moisture content, dead moisture of extinction, and live herbaceous surface-area-to-volume-ratio. In the following formula, M_f^{lh} is the live herbaceous moisture content.

$$\text{FractionGreen} = \begin{cases} 0 & M_f^{lh} \leq 0.3 \\ 1 & M_f^{lh} \geq 1.2 \\ \frac{M_f^{lh}}{0.9} - \frac{1}{3} & \text{else} \end{cases}$$

$$\text{FractionCured} = 1 - \text{FractionGreen}$$

```
(defn add-dynamic-fuel-loading
  [{:keys [number M_x M_f w_o sigma] :as fuel-model}]
  (let [number               (double number)
        live-herbaceous-load (-> w_o :live :herbaceous double)]
    (if (and (> number 100) (pos? live-herbaceous-load))
      ;; dynamic fuel model
      (let [fraction-green (max 0.0 (min 1.0 (- (/ (-> M_f :live :herbaceous double) 0.9) (/ 1.0 3.0))))
            fraction-cured (- 1.0 fraction-green)]
        (-> fuel-model
            (assoc-in [:M_f   :dead :herbaceous] (-> M_f :dead :1hr))
            (assoc-in [:M_x   :dead :herbaceous] (-> M_x :dead :1hr))
            (assoc-in [:w_o   :dead :herbaceous] (* live-herbaceous-load fraction-cured))
            (assoc-in [:w_o   :live :herbaceous] (* live-herbaceous-load fraction-green))
            (assoc-in [:sigma :dead :herbaceous] (-> sigma :live :herbaceous))))
      ;; static fuel model
      fuel-model)))
```

Once the dynamic fuel loading is applied, we can compute the size class weighting factors expressed in equations 53-57 in Rothermel 1972(Rothermel, 1972). For brevity, these formulas are elided from this text.

```
(defn add-weighting-factors
  [{:keys [w_o sigma rho_p] :as fuel-model}]
  (let [A_ij (map-size-class (fn [i j] (/ (* (-> sigma i ^double (j)) (-> w_o i ^double (j)))
                                          (-> rho_p i ^double (j)))))

        A_i  (size-class-sum (fn [i j] (-> A_ij i j)))

        A_T  (category-sum (fn [i] (-> A_i i)))

        f_ij (map-size-class (fn [i j] (if (pos? ^double ( A_i i))
                                         (/ (-> A_ij i ^double (j))
                                            ^double (A_i i))
                                         0.0)))

        f_i  (map-category (fn [i] (if (pos? A_T)
                                    (/ ^double (A_i i) A_T)
                                    0.0)))

        firemod-size-classes (map-size-class
                              (fn [i j] (condp <= (-> sigma i j)
                                          1200 1
                                          192  2
                                          96   3
                                          48   4
                                          16   5
                                          0    6)))

        firemod-weights (into {}
                              (for [[category size-classes] firemod-size-classes]
                                [category
                                 (apply merge-with +
                                        (for [[size-class firemod-size-class] size-classes]
                                          {firemod-size-class (get-in f_ij [category size-class])}))]))

        g_ij (map-size-class (fn [i j]
                               (let [firemod-size-class (-> firemod-size-classes i j)]
                                 (get-in firemod-weights [i firemod-size-class]))))]
    (-> fuel-model
        (assoc :f_ij f_ij)
        (assoc :f_i  f_i)
        (assoc :g_ij g_ij))))
```

5.1.2 Live Moisture of Extinction

The live moisture of extinction for each fuel model is determined from the dead fuel moisture content, the dead moisture of extinction, and the ratio of dead fuel loading to live fuel loading using Equation 88 from Rothermel 1972, adjusted according to Albini 1976 Appendix III to match the behavior of Albini's original FIREMODS library.(Rothermel, 1972, Albini, 1976) Whenever the fuel moisture content becomes greater than or equal to the moisture of extinction, a fire will no longer spread through that fuel. Here are the formulas referenced above:

$$M_x^l = \max(M_x^d, 2.9\, W' \,(1 - \frac{M_f^d}{M_x^d}) - 0.226)$$

$$W' = \frac{\sum_{c \in D} w_o^c \, e^{-138/\sigma^c}}{\sum_{c \in L} w_o^c \, e^{-500/\sigma^c}}$$

$$M_f^d = \frac{\sum_{c \in D} w_o^c \, M_f^c \, e^{-138/\sigma^c}}{\sum_{c \in D} w_o^c \, e^{-138/\sigma^c}}$$

where M_x^l is the live moisture of extinction, M_x^d is the dead moisture of extinction, D is the set of dead fuel size classes (1hr, 10hr, 100hr, herbaceous), L is the set of live fuel size classes (herbaceous, woody), w_o^c is the dry weight loading of size class c, σ^c is the surface area to volume ratio of size class c, and M_f^c is the moisture content of size class c.

```
(defn add-live-moisture-of-extinction
  "Equation 88 from Rothermel 1972 adjusted by Albini 1976 Appendix III."
  [{:keys [w_o sigma M_f M_x] :as fuel-model}]
  (let [dead-loading-factor  (->> (size-class-sum
                                   (fn [i j] (let [sigma_ij (-> sigma i j double)]
                                               (if (pos? sigma_ij)
                                                 (* (-> w_o i ^double (j))
                                                    (Math/exp (/ -138.0 sigma_ij)))
                                                 0.0))))
                                  :dead
                                  double)
        live-loading-factor  (->> (size-class-sum
                                   (fn [i j] (let [sigma_ij (-> sigma i j double)]
                                               (if (pos? sigma_ij)
                                                 (* (-> w_o i ^double (j))
                                                    (Math/exp (/ -500.0 sigma_ij)))
                                                 0.0))))
                                  :live
                                  double)
        dead-moisture-factor (->> (size-class-sum
                                   (fn [i j] (let [sigma_ij (-> sigma i j double)]
                                              (if (pos? sigma_ij)
                                                (* (-> w_o i ^double (j))
                                                   (Math/exp (/ -138.0 sigma_ij))
                                                   (-> M_f i ^double (j)))
                                                0.0))))
                                  :dead
                                  double)
        ^double
        dead-to-live-ratio   (when (pos? live-loading-factor)
                               (/ dead-loading-factor live-loading-factor))
        dead-fuel-moisture   (if (pos? dead-loading-factor)
                               (/ dead-moisture-factor dead-loading-factor)
                               0.0)
        M_x-dead             (-> M_x :dead :1hr double)
        M_x-live             (if (pos? live-loading-factor)
                               (max M_x-dead
                                    (- (* 2.9
                                          dead-to-live-ratio
                                          (- 1.0 (/ dead-fuel-moisture M_x-dead)))
                                       0.226))
                               M_x-dead)]
    (-> fuel-model
        (assoc-in [:M_x :live :herbaceous] M_x-live)
        (assoc-in [:M_x :live :woody]      M_x-live))))

(defn moisturize
  [fuel-model fuel-moisture]
  (-> fuel-model
      (assoc :M_f fuel-moisture)
      (assoc-in [:M_f :dead :herbaceous] 0.0)
      (add-dynamic-fuel-loading)
      (add-weighting-factors)
      (add-live-moisture-of-extinction)))
```

This concludes our coverage of fuel models and and fuel moisture.

5.2 Surface Fire Formulas

To simulate fire behavior in as similar a way as possible to the US government-sponsored fire models (e.g., FARSITE, FlamMap, FPA, BehavePlus), we adopt the surface fire spread and reaction intensity formulas from Rothermel's 1972 publication "A Mathematical Model for Predicting Fire Spread in Wildland Fuels".(Rothermel, 1972)

Very briefly, the surface rate of spread of a fire's leading edge R is described by the following formula:

$$R = \frac{I_R \, \xi \, (1 + \phi_W + \phi_S)}{\rho_b \, \epsilon \, Q_{ig}}$$

where these terms have the meanings shown in Table 3.

For a full description of each of the subcomponents of Rothermel's surface fire spread rate equation, see the Rothermel 1972 reference above. In addition to applying the base Rothermel equations, GridFire

Table 3: Inputs to Rothermel's surface fire rate of spread equation

Term	Meaning
R	surface fire spread rate
I_R	reaction intensity
ξ	propagating flux ratio
ϕ_W	wind coefficient
ϕ_S	slope factor
ρ_b	oven-dry fuel bed bulk density
ϵ	effective heating number
Q_{ig}	heat of preignition

reduces the spread rates for all of the Scott & Burgan 40 fuel models of the grass subgroup (101-109) by 50%. This addition was originally suggested by Chris Lautenberger of REAX Engineering.

For efficiency, the surface fire spread equation given above is computed first without introducing the effects of wind and slope ($\phi_W = \phi_S = 0$).

```
(ns gridfire.surface-fire
  (:require [gridfire.fuel-models :refer [map-category map-size-class
                                          category-sum size-class-sum]]))

(defn grass-fuel-model?
  [^long number]
  (and (> number 100) (< number 110)))

(defn rothermel-surface-fire-spread-no-wind-no-slope
  "Returns the rate of surface fire spread in ft/min and the reaction
   intensity (i.e., amount of heat output) of a fire in Btu/ft^2*min
   given a map containing these keys:
   - number [fuel model number]
   - delta [fuel depth (ft)]
   - w_o [ovendry fuel loading (lb/ft^2)]
   - sigma [fuel particle surface-area-to-volume ratio (ft^2/ft^3)]
   - h [fuel particle low heat content (Btu/lb)]
   - rho_p [ovendry particle density (lb/ft^3)]
   - S_T [fuel particle total mineral content (lb minerals/lb ovendry wood)]
   - S_e [fuel particle effective mineral content (lb silica-free minerals/lb ovendry wood)]
   - M_x [moisture content of extinction (lb moisture/lb ovendry wood)]
   - M_f [fuel particle moisture content (lb moisture/lb ovendry wood)]
   - f_ij [percent of load per size class (%)]
   - f_i [percent of load per category (%)]
   - g_ij [percent of load per size class from Albini_1976_FIREMOD, page 20]"
  [{:keys [number delta w_o sigma  h rho_p S_T S_e M_x  M_f f_ij f_i g_ij]}]
  (let [number     (long number)
        delta      (double delta)
        S_e_i      (size-class-sum (fn [i j] (* (-> f_ij i ^double (j)) (-> S_e i ^double (j)))))

        ;; Mineral damping coefficient
        eta_S_i    (map-category (fn [i] (let [^double S_e_i (-> S_e_i i)]
                                           (if (pos? S_e_i)
                                             (/ 0.174 (Math/pow S_e_i 0.19))
                                             1.0))))

        M_f_i      (size-class-sum (fn [i j] (* (-> f_ij i ^double (j)) (-> M_f i ^double (j)))))

        M_x_i      (size-class-sum (fn [i j] (* (-> f_ij i ^double (j)) (-> M_x i ^double (j)))))

        r_M_i      (map-category (fn [i] (let [^double M_f (-> M_f_i i)
                                               ^double M_x (-> M_x_i i)]
                                           (if (pos? M_x)
                                             (min 1.0 (/ M_f M_x))
                                             1.0))))
```

```
;; Moisture damping coefficient
eta_M_i     (map-category (fn [i] (+ 1.0
                                   (* -2.59 (-> r_M_i ^double (i)))
                                   (* 5.11 (Math/pow (-> r_M_i i) 2))
                                   (* -3.52 (Math/pow (-> r_M_i i) 3)))))

h_i         (size-class-sum (fn [i j] (* (-> f_ij i ^double (j)) (-> h i ^double (j)))))

;; Net fuel loading (lb/ft^2)
W_n_i       (size-class-sum (fn [i j] (* (-> g_ij i ^double (j))
                                         (-> w_o i ^double (j))
                                         (- 1.0 (-> S_T i ^double (j))))))

beta_i      (size-class-sum (fn [i j] (/ (-> w_o i ^double (j)) (-> rho_p i ^double (j)))))

;; Packing ratio
beta        (if (pos? delta)
              (/ (category-sum (fn [i] (-> beta_i ^double (i)))) delta)
              0.0)

sigma'_i    (size-class-sum (fn [i j] (* (-> f_ij i ^double (j)) (-> sigma i ^double (j)))))

sigma'      (category-sum (fn [i] (* (-> f_i ^double (i)) (-> sigma'_i ^double (i)))))

;; Optimum packing ratio
beta_op     (if (pos? sigma')
              (/ 3.348 (Math/pow sigma' 0.8189))
              1.0)

;; Albini 1976 replaces (/ 1 (- (* 4.774 (Math/pow sigma' 0.1)) 7.27))
A           (if (pos? sigma')
              (/ 133.0 (Math/pow sigma' 0.7913))
              0.0)

;; Maximum reaction velocity (1/min)
Gamma'_max (/ (Math/pow sigma' 1.5)
               (+ 495.0 (* 0.0594 (Math/pow sigma' 1.5))))

;; Optimum reaction velocity (1/min)
Gamma'      (* Gamma'_max
               (Math/pow (/ beta beta_op) A)
               (Math/exp (* A (- 1.0 (/ beta beta_op)))))

;; Reaction intensity (Btu/ft^2*min)
I_R         (* Gamma' (category-sum (fn [i] (* ^double (W_n_i i) ^double (h_i i)
                                               ^double (eta_M_i i) ^double (eta_S_i i)))))

;; Propagating flux ratio
xi          (/ (Math/exp (* (+ 0.792 (* 0.681 (Math/pow sigma' 0.5)))
                            (+ beta 0.1)))
               (+ 192.0 (* 0.2595 sigma')))

E           (* 0.715 (Math/exp (* -3.59 (/ sigma' 10000.0))))

B           (* 0.02526 (Math/pow sigma' 0.54))

C           (* 7.47 (Math/exp (* -0.133 (Math/pow sigma' 0.55))))

;; Derive wind factor
get-phi_W   (fn ^double [^double midflame-wind-speed]
              (if (and (pos? beta) (pos? midflame-wind-speed))
                (-> midflame-wind-speed
                    (Math/pow B)
                    (* C)
                    (/ (Math/pow (/ beta beta_op) E)))
                0.0))

;; Derive wind speed from wind factor
```

```
        get-wind-speed (fn [^double phi_W]
                         (-> phi_W
                             (* (Math/pow (/ beta beta_op) E))
                             ^double (/ C)
                             (Math/pow (/ 1.0 B))))

        ;; Derive slope factor
        get-phi_S  (fn [^double slope]
                     (if (and (pos? beta) (pos? slope))
                       (* 5.275 (Math/pow beta -0.3) (Math/pow slope 2.0))
                       0.0))

        ;; Heat of preignition (Btu/lb)
        Q_ig       (map-size-class (fn [i j] (+ 250.0 (* 1116.0 (-> M_f i ^double (j))))))

        foo_i      (size-class-sum (fn [i j] (let [^double sigma_ij (-> sigma i j)
                                                   ^double Q_ig_ij  (-> Q_ig  i j)]
                                               (if (pos? sigma_ij)
                                                 (* (-> f_ij i ^double (j))
                                                    (Math/exp (/ -138 sigma_ij))
                                                    Q_ig_ij)
                                                 0.0))))

        rho_b_i    (size-class-sum (fn [i j] (-> w_o i j)))

        ;; Ovendry bulk density (lb/ft^3)
        rho_b      (if (pos? delta)
                     (/ (category-sum (fn [i] (-> rho_b_i i))) delta)
                     0.0)

        rho_b-epsilon-Q_ig (* rho_b (category-sum (fn [i] (* (-> f_i ^double (i)) (-> foo_i ^double (i))))))

        ;; Surface fire spread rate (ft/min)
        R          (if (pos? rho_b-epsilon-Q_ig)
                     (/ (* I_R xi) rho_b-epsilon-Q_ig)
                     0.0)

        ;; Addition proposed by Chris Lautenberger (REAX 2015)
        spread-rate-multiplier (if (grass-fuel-model? number) 0.5 1.0)]

    {:spread-rate        (* R spread-rate-multiplier)
     :reaction-intensity I_R
     :residence-time     (/ 384.0 sigma')
     :get-phi_W          get-phi_W
     :get-phi_S          get-phi_S
     :get-wind-speed     get-wind-speed}))
```

Later, this no-wind-no-slope value is used to compute the maximum spread rate and direction for the leading edge of the surface fire under analysis. Since Rothermel's original equations assume that the wind direction and slope are aligned, the effects of cross-slope winds must be taken into effect. Like Morais' HFire system, GridFire implements the vector addition procedure defined in Rothermel 1983 that combines the wind-only and slope-only spread rates independently to calculate the effective fire spread direction and magnitude.(Peterson et al., 2011, 2009, Morais, 2001, Rothermel, 1983)

A minor wrinkle is introduced when putting these calculations into practice because Rothermel's formulas all expect a measure of midflame wind speed. However, wind speed data is often collected at a height 20 feet above either unsheltered ground or a tree canopy layer if present. To convert this 20-ft wind speed to the required midflame wind speed value, GridFire uses the **wind adjustment factor** formula from Albini & Baughman 1979, parameterized as in BehavePlus, FARSITE, FlamMap, FSPro, and FPA according to Andrews 2012(Albini and Baughman, 1979, Andrews et al., 2012). This formula is shown below:

$$WAF = \begin{cases} \frac{1.83}{\ln(\frac{20.0+0.36FBD}{0.13FBD})} & CC = 0 \\ \frac{0.555}{\sqrt{(CH(CC/300.0))}\ln(\frac{20+0.36CH}{0.13CH})} & CC > 0 \end{cases}$$

where WAF is the unitless wind adjustment factor, FBD is the fuel bed depth in feet, CH is the canopy height in ft, and CC is the canopy cover percentage (0-100).

```
(defn wind-adjustment-factor
  "ft ft 0-100"
  ^double
  [^double fuel-bed-depth ^double canopy-height ^double canopy-cover]
  (cond
    ;; sheltered: equation 2 based on CC and CH, CR=1 (Andrews 2012)
    (and (pos? canopy-cover)
         (pos? canopy-height))
    (/ 0.555 (* (Math/sqrt (* (/ canopy-cover 300.0) canopy-height))
                (Math/log (/ (+ 20.0 (* 0.36 canopy-height)) (* 0.13 canopy-height)))))

    ;; unsheltered: equation 6 H_F = H (Andrews 2012)
    (pos? fuel-bed-depth)
    (/ 1.83 (Math/log (/ (+ 20.0 (* 0.36 fuel-bed-depth)) (* 0.13 fuel-bed-depth))))

    ;; non-burnable fuel model
    :otherwise
    0.0))

(defn wind-adjustment-factor-elmfire
  "ft m 0-1"
  ^double
  [^double fuel-bed-depth ^double canopy-height ^double canopy-cover]
  (cond
    ;; sheltered WAF
    (and (pos? canopy-cover)
         (pos? canopy-height))
    (* (/ 1.0 (Math/log (/ (+ 20.0 (* 0.36 (/ canopy-height 0.3048)))
                           (* 0.13 (/ canopy-height 0.3048)))))
       (/ 0.555 (Math/sqrt (* (/ canopy-cover 3.0) (/ canopy-height 0.3048)))))

    ;; unsheltered WAF
    (pos? fuel-bed-depth)
    (* (/ (+ 1.0 (/ 0.36 1.0))
          (Math/log (/ (+ 20.0 (* 0.36 fuel-bed-depth))
                       (* 0.13 fuel-bed-depth))))
       (- (Math/log (/ (+ 1.0 0.36) 0.13)) 1.0))

    ;; non-burnable fuel model
    :otherwise
    0.0))
```

The midflame wind speed that would be required to produce the combined spread rate in a no-slope scenario is termed the effective windspeed U_{eff}. Following the recommendations given in Appendix III of Albini 1976, these midflame wind speeds are all limited to $0.9 I_R$.(Albini, 1976)

Next, the effective wind speed is used to compute the length to width ratio $\frac{L}{W}$ of an ellipse that approximates the fire front using equation 9 from Rothermel 1991.(Rothermel, 1991) This length to width ratio is then converted into an eccentricity measure of the ellipse using equation 8 from Albini and Chase 1980.(Albini and Chase, 1980) Finally, this eccentricity E is used to project the maximum spread rate to any point along the fire front. Here are the formulas used:

$$\frac{L}{W} = 1 + 0.002840909\, U_{\mathrm{eff}}\, \mathrm{EAF}$$
$$E = \frac{\sqrt{(\frac{L}{W})^2 - 1}}{\frac{L}{W}}$$
$$R_\theta = R_{\max}\left(\frac{1-E}{1-E\cos\theta}\right)$$

where θ is the angular offset from the direction of maximum fire spread, $\mathrm{R}_{\max}$ is the maximum spread rate, R_θ is the spread rate in direction θ, and EAF is the ellipse adjustment factor, a term introduced by Marco Morais and Seth Peterson in their HFire work that can be increased or decreased to make the fire shape more elliptical or circular respectively.(Peterson et al., 2009)

Note: The coefficient 0.002840909 in the $\frac{L}{W}$ formula is in units of min/ft. The original equation from Rothermel 1991 used 0.25 in units of hr/mi, so this was converted to match GridFire's use of ft/min for U_{eff}.

```
(defn almost-zero? [^double x]
  (< (Math/abs x) 0.000001))

(defn degrees-to-radians ^double
  [^double degrees]
  (/ (* degrees Math/PI) 180.0))

(defn radians-to-degrees
  ^double
  [^double radians]
  (/ (* radians 180.0) Math/PI))

(defn scale-spread-to-max-wind-speed
  [{:keys [effective-wind-speed max-spread-direction] :as spread-properties}
   ^double spread-rate max-wind-speed  ^double phi-max]
  (let [effective-wind-speed (double effective-wind-speed)
        max-spread-direction (double max-spread-direction)]
    (if (> effective-wind-speed ^double max-wind-speed)
      {:max-spread-rate      (* spread-rate (+ 1.0 phi-max))
       :max-spread-direction max-spread-direction
       :effective-wind-speed max-wind-speed}
      spread-properties)))

(defn add-eccentricity
  [{:keys [effective-wind-speed] :as spread-properties} ellipse-adjustment-factor]
  (let [effective-wind-speed (double effective-wind-speed)
        length-width-ratio (+ 1.0 (* 0.002840909
                                     effective-wind-speed
                                     ^double ellipse-adjustment-factor))
        eccentricity       (/ (Math/sqrt (- (Math/pow length-width-ratio 2.0) 1.0))
                              length-width-ratio)]
    (assoc spread-properties :eccentricity eccentricity)))

(defn smallest-angle-between ^double
  [^double theta1 ^double theta2]
  (let [angle (Math/abs (- theta1 theta2))]
    (if (> angle 180.0)
      (- 360.0 angle)
      angle)))

(defn rothermel-surface-fire-spread-max
  "Note: fire ellipse adjustment factor, < 1.0 = more circular, > 1.0 = more elliptical"
  [{:keys [spread-rate reaction-intensity get-phi_W get-phi_S get-wind-speed]}
   midflame-wind-speed wind-from-direction slope aspect ellipse-adjustment-factor]
  (let [^double phi_W             (get-phi_W midflame-wind-speed)
        ^double phi_S             (get-phi_S slope)
        ^double slope-direction   (mod (+ ^double aspect 180.0) 360.0)
        ^double wind-to-direction (mod (+ ^double wind-from-direction 180.0) 360.0)
        max-wind-speed            (* 0.9 ^double reaction-intensity)
        ^double phi-max           (get-phi_W max-wind-speed)
        spread-rate               (double spread-rate)]
    (->
     (cond (and (almost-zero? midflame-wind-speed) (almost-zero? slope))
           ;; no wind, no slope
           {:max-spread-rate      spread-rate
            :max-spread-direction 0.0
            :effective-wind-speed 0.0}

           (almost-zero? slope)
           ;; wind only
           {:max-spread-rate      (* spread-rate (+ 1.0 phi_W))
            :max-spread-direction wind-to-direction
            :effective-wind-speed midflame-wind-speed}

           (almost-zero? midflame-wind-speed)
           ;; slope only
           {:max-spread-rate      (* spread-rate (+ 1.0 phi_S))
            :max-spread-direction slope-direction
            :effective-wind-speed (get-wind-speed phi_S)}

           (< (smallest-angle-between wind-to-direction slope-direction) 15.0)
           ;; wind blows (within 15 degrees of) upslope
           {:max-spread-rate      (* spread-rate (+ 1.0 phi_W phi_S))
            :max-spread-direction slope-direction
            :effective-wind-speed (get-wind-speed (+ phi_W phi_S))}

           :else
           ;; wind blows across slope
           (let [slope-magnitude    (* spread-rate phi_S)
                 wind-magnitude     (* spread-rate phi_W)
                 difference-angle   (degrees-to-radians
                                     (mod (- wind-to-direction slope-direction) 360.0))
                 x                  (+ slope-magnitude
                                       (* wind-magnitude (Math/cos difference-angle)))
                 y                  (* wind-magnitude (Math/sin difference-angle))
                 combined-magnitude (Math/sqrt (+ (* x x) (* y y)))]
             (if (almost-zero? combined-magnitude)
               {:max-spread-rate      spread-rate
                :max-spread-direction 0.0
                :effective-wind-speed 0.0}
               (let [max-spread-rate      (+ spread-rate combined-magnitude)
                     phi-combined         (- (/ max-spread-rate spread-rate) 1.0)
                     offset               (radians-to-degrees
                                           (Math/asin (/ (Math/abs y) combined-magnitude)))
                     offset'              (if (>= x 0.0)
                                            (if (>= y 0.0)
                                              offset
                                              (- 360.0 offset))
                                            (if (>= y 0.0)
                                              (- 180.0 offset)
                                              (+ 180.0 offset)))
                     max-spread-direction (mod (+ slope-direction offset') 360.0)
                     effective-wind-speed (get-wind-speed phi-combined)]
                 {:max-spread-rate      max-spread-rate
                  :max-spread-direction max-spread-direction
                  :effective-wind-speed effective-wind-speed}))))
     (scale-spread-to-max-wind-speed spread-rate max-wind-speed phi-max)
     (add-eccentricity ellipse-adjustment-factor))))

(defn rothermel-surface-fire-spread-any ^double
  [{:keys [max-spread-rate max-spread-direction eccentricity]} spread-direction]
  (let [max-spread-rate      (double max-spread-rate)
        max-spread-direction (double max-spread-direction)
        eccentricity         (double eccentricity)
        theta                (smallest-angle-between max-spread-direction spread-direction)]
    (if (or (almost-zero? eccentricity) (almost-zero? theta))
      max-spread-rate
      (* max-spread-rate (/ (- 1.0 eccentricity)
                            (- 1.0 (* eccentricity
                                      (Math/cos (degrees-to-radians theta))))))))
```

Using these surface fire spread rate and reaction intensity values, we next calculate fire intensity values by applying Anderson's flame depth formula and Byram's fire line intensity and flame length equations as described below.(Anderson, 1969, Byram, 1959)

$$t = \frac{384}{\sigma}$$
$$D = Rt$$
$$I = \frac{I_R D}{60}$$
$$L = 0.45(I)^{0.46}$$

where σ is the weighted sum by size class of the fuel model's surface area to volume ratio in $\mathrm{ft}^2/\mathrm{ft}^3$, t is the residence time in minutes, R is the surface fire spread rate in ft/min, D is the flame depth in ft, I_R is the reaction intensity in $\mathrm{Btu}/\mathrm{ft}^2/\mathrm{min}$, I is the fire line intensity in Btu/ft/s, and L is the flame length in ft.

```
(defn anderson-flame-depth
  "Returns the depth, or front-to-back distance, of the actively flaming zone
   of a free-spreading fire in ft given:
   - spread-rate (ft/min)
   - residence-time (min)"
  ^double
  [^double spread-rate ^double residence-time]
  (* spread-rate residence-time))

(defn byram-fire-line-intensity
  "Returns the rate of heat release per unit of fire edge in Btu/ft*s given:
   - reaction-intensity (Btu/ft^2*min)
   - flame-depth (ft)"
  ^double
  [^double reaction-intensity ^double flame-depth]
  (/ (* reaction-intensity flame-depth) 60.0))

(defn byram-flame-length
  "Returns the average flame length in ft given:
   - fire-line-intensity (Btu/ft*s)"
  ^double
  [^double fire-line-intensity]
  (* 0.45 (Math/pow fire-line-intensity 0.46)))
```

This concludes our coverage of the surface fire behavior equations implemented in GridFire. In Section 5.4, these formulas will be translated from one-dimension to two-dimensional spread on a raster grid. Before we move on to that, however, the following section explains how crown fire behavior metrics are incorporated into our model.

5.3 Crown Fire Formulas

In order to incorporate the effects of crown fire behavior, GridFire includes the crown fire initiation routine from Van Wagner 1977.(Wagner, 1977) According to this approach, there are two threshold values (*critical* *intensity* and *critical spread rate*) that must be calculated in order to determine whether a fire will become an active or passive crown fire or simply remain a surface fire. The formulas for these thresholds are as follows:

$$H = 460 + 2600M^f$$
$$I^* = (0.01\, Z_b\, H)^{1.5}$$
$$R^* = \frac{3.0}{B_m}$$

where H is the heat of ignition for the herbaceous material in the canopy in kJ/kg, M^f is the foliar moisture content in lb moisture/lb ovendry weight, Z_b is the canopy base height in meters, I^* is the critical intensity in kW/m, B_m is the crown bulk density in kg/m^3, and R^* is the critical spread rate in m/min.

If the canopy cover is greater than 40% and the surface fire line intensity is greater than the critical intensity ($I > I^*$), then crown fire initiation occurs.

```
(ns gridfire.crown-fire
  (:require [gridfire.conversion :as convert]))

(defn van-wagner-crown-fire-initiation?
  "- canopy-cover (0-100 %)
   - canopy-base-height (ft)
   - foliar-moisture (lb moisture/lb ovendry weight)
   - fire-line-intensity (Btu/ft*s)"
  [^double canopy-cover ^double canopy-base-height ^double foliar-moisture ^double fire-line-intensity]
  (and (> canopy-cover 40.0)
       (-> (+ 460.0 (* 2600.0 foliar-moisture)) ;; heat-of-ignition = kJ/kg
           (* 0.01 (convert/ft->m canopy-base-height))
           (Math/pow 1.5) ;; critical-intensity = kW/m
           (convert/kW-m->Btu-ft-s)
           (< fire-line-intensity))))
```

If crowning occurs, then the active and passive crown fire spread rates are calculated from the formulas given in Cruz 2005.(Cruz et al., 2005)

$$\text{CROS}_A = 11.02\, U_{10m}^{0.90}\, B_m^{0.19}\, e^{-0.17\,\text{EFFM}}$$
$$\text{CROS}_P = \text{CROS}_A\, e^{\frac{-\text{CROS}_A}{R^*}}$$

where CROS_A is the active crown fire spread rate in m/min, U_{10m} is the 10 meter windspeed in km/hr, B_m is the crown bulk density in kg/m^3, EFFM is the estimated fine fuel moisture as a percent (0-100), and CROS_P is the passive crown fire spread rate in m/min.

If the active crown fire spread rate is greater than the critical spread rate ($\text{CROS}_A > R^*$), then the crown fire will be active, otherwise passive.

```
(defn cruz-crown-fire-spread
  "Returns spread-rate in ft/min given:
   - wind-speed-20ft (mph)
   - crown-bulk-density (lb/ft^3)
   - estimated-fine-fuel-moisture (-> M_f :dead :1hr) (0-1)"
  [wind-speed-20ft ^double crown-bulk-density ^double estimated-fine-fuel-moisture]
  (let [wind-speed-10m               (/ (convert/mph->km-hr wind-speed-20ft) 0.87) ;; km/hr
        crown-bulk-density           (convert/lb-ft3->kg-m3 crown-bulk-density) ;; kg/m^3
        estimated-fine-fuel-moisture (* 100.0 estimated-fine-fuel-moisture)
        active-spread-rate           (* 11.02
                                          (Math/pow wind-speed-10m 0.90)
                                          (Math/pow crown-bulk-density 0.19)
                                          (Math/exp (* -0.17 estimated-fine-fuel-moisture)))
        ;; m/min
        critical-spread-rate         (/ 3.0 crown-bulk-density) ;; m/min
        criteria-for-active-crowning (/ active-spread-rate critical-spread-rate)]
    (if (> active-spread-rate critical-spread-rate)
      [:active-crown (convert/m->ft active-spread-rate)]
      [:passive-crown (convert/m->ft (* active-spread-rate (Math/exp (- criteria-for-active-crowning))))])))
```

Once the crown fire spread rate is determined, the crown fire line intensity and flame lengths may be derived using the following formulas:

$$I_c = \frac{R_c B(Z - Z_b)h}{60}$$
$$L_c = 0.45(I + I_c)^{0.46}$$

where I_c is the crown fire line intensity in Btu/ft/s, R_c is the crown fire spread rate (either CROS$_A$ or CROS$_P$) in ft/min, B is the crown bulk density in lb/ft^3, Z is the canopy height in ft, Z_b is the canopy base height in ft, h is the fuel model heat of combustion (generally 8000 Btu/lb), L_c is the crown flame length in ft, and I is the surface fire line intensity in Btu/ft/s.

```
;; heat of combustion is h from the fuel models (generally 8000 Btu/lb)
(defn crown-fire-line-intensity
  "(ft/min * lb/ft^3 * ft * Btu/lb)/60 = (Btu/ft*min)/60 = Btu/ft*s"
  [crown-spread-rate crown-bulk-density canopy-height canopy-base-height heat-of-combustion]
  (/ (* ^double crown-spread-rate
        ^double crown-bulk-density
        (- ^double canopy-height ^double canopy-base-height)
        ^double heat-of-combustion)
     60.0))

(defn crown-fire-line-intensity-elmfire ;; kW/m
  [surface-fire-line-intensity crown-spread-rate crown-bulk-density
   canopy-height canopy-base-height]
  (let [heat-of-combustion 18000] ;; kJ/m^2
    (+ ^double surface-fire-line-intensity ;; kW/m
       (/ (* 0.3048 ;; m/ft
             ^double crown-spread-rate ;; ft/min
             ^double crown-bulk-density ;; kg/m^3
             (- ^double canopy-height ^double canopy-base-height) ;; m
             heat-of-combustion) ;; kJ/kg
          60.0)))) ;; s/min
```

As with surface fire spread, the wind speed (this time the 20-ft wind speed in mph U_{20}) is used to compute the length to width ratio $\frac{L}{W}$ of an ellipse that approximates the crown fire front using equation 9 from Rothermel 1991.(Rothermel, 1991) This length to width ratio is then converted into an eccentricity measure of the ellipse using equation 8 from Albini and Chase 1980.(Albini and Chase, 1980) Finally, this eccentricity E is used to project the maximum spread rate to any point along the fire front. Here are the formulas used:

$$\frac{L}{W} = 1 + 0.125\,U_{20}\,\text{EAF}$$

$$E = \frac{\sqrt{(\frac{L}{W})^2 - 1}}{\frac{L}{W}}$$

$$R_\theta = R_{\max}\left(\frac{1 - E}{1 - E\cos\theta}\right)$$

where θ is the angular offset from the direction of maximum fire spread, R_{max} is the maximum spread rate, R_θ is the spread rate in direction θ, and EAF is the ellipse adjustment factor, a term introduced by Marco Morais and Seth Peterson in their HFire work that can be increased or decreased to make the fire shape more elliptical or circular respectively.(Peterson et al., 2009)

```
(defn crown-fire-eccentricity
  "mph"
  ^double
  [^double wind-speed-20ft ^double ellipse-adjustment-factor]
  (let [length-width-ratio (+ 1.0 (* 0.125
                                     wind-speed-20ft
                                     ellipse-adjustment-factor))]
    (/ (Math/sqrt (- (Math/pow length-width-ratio 2.0) 1.0))
       length-width-ratio)))

(defn elmfire-length-to-width-ratio
  "true/false mph int>0 ft/min
   Crown L/W = min(1.0 + 0.125*U20_mph, L/W_max)
   Surface L/W = 0.936*e^(0.2566*Ueff_mph) + 0.461*e^(-0.1548*Ueff_mph) - 0.397"
  ^double
  [crown-fire? ^double wind-speed-20ft ^double max-length-to-width-ratio ^double effective-wind-speed]
  (if crown-fire?
    (min (+ 1.0 (* 0.125 wind-speed-20ft)) max-length-to-width-ratio)
    (min (+ (* 0.936 (Math/exp (/ (* 0.2566 effective-wind-speed 60.0) 5280.0)))
            (* 0.461 (Math/exp (/ (* -0.1548 effective-wind-speed 60.0) 5280.0)))
            -0.397)
         8.0)))
```

This concludes our discussion of the crown fire behavior formulas used in GridFire.

5.4 Fire Spread on a Raster Grid

Although Rothermel's spread rate formula provides some useful insight into how quickly a fire's leading edge may travel, it offers no specific mechanism for simulating fire movement in two or more dimensions. Therefore, when attempting to use the Rothermel equations in any spatial analysis, one must begin by choosing a model of space and then decide how best to employ the spread rate equations along each possible burn trajectory.

In GridFire, SIG adopted a raster grid view of space so as to reduce the potentially exponential complexity of modeling a fractal shape (i.e., fire front) at high resolutions using vector approximation. This also provided the practical benefit of being able to work directly with widely used raster datasets, such as LANDFIRE, without a geometric lookup step or *a priori* translation to vector space.

In simulation tests versus FARSITE on several historical California fires, Marco Morais wrote that he saw similarly accurate results from both his HFire model and from FARSITE but experienced several orders of magnitude improvement in runtime efficiency.(Peterson et al., 2011, 2009, Morais, 2001) His explanation for this phenomenon was in the same vein as that described above, namely, that it was FARSITE's choice of vector space that slowed it down versus the faster raster-based HFire system.

Taking a cue from HFire's success in this regard, GridFire has adopted HFire's two-dimensional spread algorithm, called the *method of adaptive timesteps and fractional distances.* (Peterson et al., 2011, 2009, Morais, 2001) The following pseudo-code lays out the steps taken in this procedure:

1. Inputs

    (a) Read in the values shown in Table 4.

Table 4: Inputs to SIG's raster-based fire behavior model

Value	Units	Type
max-runtime	minutes	double
cell-size	feet	double
elevation-matrix	feet	core.matrix 2D double array
slope-matrix	vertical feet/horizontal feet	core.matrix 2D double array
aspect-matrix	degrees clockwise from north	core.matrix 2D double array
fuel-model-matrix	fuel model numbers 1-256	core.matrix 2D double array
canopy-height-matrix	feet	core.matrix 2D double array
canopy-base-height-matrix	feet	core.matrix 2D double array
crown-bulk-density-matrix	lb/ft^3	core.matrix 2D double array
canopy-cover-matrix	0-100	core.matrix 2D double array
wind-speed-20ft	miles/hour	double
wind-from-direction	degrees clockwise from North	double
fuel-moisture	%	map of doubles per fuel size class
foliar-moisture	%	double
ellipse-adjustment-factor	< 1.0 = circle, > 1.0 = ellipse	double
initial-ignition-site	point represented as [row col]	vector

2. Initialization

    (a) Verify that **initial-ignition-site** and at least one of its neighboring cells has a burnable fuel model (not 91-99). Otherwise, terminate the simulation, indicating that no fire spread is possible.

    (b) Create three new matrices, called **fire-spread-matrix**, **flame-length-matrix**, and **fire-line-intensity-matrix**. All three are initialized to zero except for a value of 1 at the **initial-ignition-site**.

    (c) Set **global-clock** to 0. This will track the amount of time that has passed since the initial ignition in minutes.

    (d) Create a new hash-map, called **ignited-cells**, which maps the **initial-ignition-site** to a set of trajectories into each of its burnable neighbors. See "Computing Burn Trajectories" below for the steps used in this procedure.

3. Computing Burn Trajectories

    (a) Look up the fuel model, slope, aspect, canopy height, canopy base height, crown bulk density, and canopy cover associated with the ignited cell in the input matrices.

    (b) Calculate the dead herbaceous size class parameters, live moisture of extinction, and size class weighting factors for this fuel model.

(c) Use the Rothermel equations to calculate the minimum surface rate of spread (i.e., wind = slope = 0) leaving this cell.

(d) Compute Albini and Baughman's wind adjustment factor for this cell using the fuel bed depth, canopy height, and canopy cover. Multiply this value by the 20-ft wind speed to derive the local midflame wind speed.

(e) Calculate the maximum surface rate of spread (and bearing) originating from this cell using the Rothermel equations and taking into account the effects of downhill and cross-slope winds as described in Rothermel 1983.

(f) Use the Cruz formulas to calculate the maximum crown fire spread rate from the 20-ft wind speed, crown bulk density, and dead 1-hr fuel moisture.

(g) Determine the surface and crown elliptical eccentricities by calculating their length-to-width ratios using the equations from Rothermel 1991.

(h) For each burnable neighboring cell:

   i. Use the eccentricity values to determine the possible surface and crown rates of spread into it from the ignited cell.

   ii. Compute Byram's surface fire line intensity and Rothermel's crown intensity from these spread rates.

   iii. Apply Van Wagner's crown initiation model to determine if the fire will be a passive or active crown fire or remain a surface fire.

   iv. In the surface fire case, the spread rate into this neighbor will simply be the surface spread rate calculated above. The fire line intensity is the surface fire line intensity, and the flame length is calculated from this intensity value using Byram's relation.

   v. In the case of a crown fire, the spread rate into this neighbor will be the maximum of the surface and crown spread rates. The fire line intensity is the sum of the surface and crown intensities, and the flame length is once again computed from Byram's relation.

   vi. Store this neighboring cell, the bearing to it from the ignited cell, and the spread rate, fire line intensity, and flame length values computed above in a burn trajectory record. Also include the terrain (e.g., 3d) distance between this cell and the ignited cell. Finally, set its **fractional-distance** value to be 0, or in the event that this bearing matches an overflow bearing from a previous iteration, set it to the **overflow-heat** value.

(i) Return a collection of burn trajectory records, one per burnable neighboring cell.

4. Main Loop

(a) If **global-clock** has not yet reached **max-runtime** and **ignited-cells** is not empty, proceed to 4.(b). Otherwise, jump to 5.(a).

(b) The timestep for this iteration of the model is calculated by dividing **cell-size** by the maximum spread rate into any cell from those cells in the **ignited-cells** map. As spread rates increase, the timesteps grow shorter and the model takes more iterations to complete. Similarly, the model has longer timesteps and takes less iterations as spread rates decrease. This is called the *method of adaptive timesteps.*

(c) If the timestep calculated in 4.(b) would cause the **global-clock** to exceed the max-runtime, then the timestep is set to the difference between **max-runtime** and **global-clock**.

(d) For each burn trajectory in **ignited-cells**:

   i. Multiply the spread rate (ft/min) by the timestep (min) to get the distance traveled by the fire (ft) along this path during this iteration.
   ii. Divide this distance traveled by the terrain distance between these two cells to get the new spread fraction $\in [0, 1]$ and increment the **fractional-distance** associated with the trajectory by this value.
   iii. If the new **fractional-distance** is greater than or equal to 1, append this updated burn trajectory record to a list called **ignition-events**.

(e) If more than one trajectory in **ignition-events** shares the same target cell, retain only the trajectory with the largest **fractional-distance** value.

(f) For each trajectory in **ignition-events**:

   i. Set the target cell's value to 1 in **fire-spread-matrix**, **flame-length** in **flame-length-matrix**, and **fire-line-intensity** in **fire-line-intensity-matrix**.
   ii. If the target cell has any burnable neighbors, append an entry to **ignited-cells**, mapping this cell to each of the burn trajectories emanating from it, which are calculated by following the steps in section "Computing Burn Trajectories" above. If its **fractional-distance** value is greater than 1, add the overflow amount above 1 to the outgoing trajectory with the same bearing along which this cell was ignited. That is, if this cell was ignited by a neighbor to the southeast, then pass any overflow heat onto the trajectory leading to the northwest.

(g) Remove any trajectories from **ignited-cells** that have as their targets any of the cells in **ignition-events**.

(h) Remove any cells from **ignited-cells** that no longer have any burnable neighbors.

(i) Increment the **global-clock** by this iteration's **timestep**.

(j) Repeat from 4.(a).

5. Outputs

   (a) Return an associative map with the fields shown in Table 5.

Table 5: Outputs from SIG's raster-based fire behavior model

Value	Units	Type
global-clock	minutes	double
initial-ignition-site	point represented as [row col]	vector
ignited-cells	list of points represented as [row col]	list of vectors
fire-spread-matrix	[0,1]	core.matrix 2D double array
flame-length-matrix	feet	core.matrix 2D double array
fire-line-intensity-matrix	Btu/ft/s	core.matrix 2D double array

```
(ns gridfire.fire-spread
  (:require [clojure.core.matrix          :as m]
            [clojure.core.reducers        :as r]
            [gridfire.common              :refer [burnable-fuel-model?
                                                  burnable?
                                                  get-fuel-moisture
                                                  fuel-moisture-from-raster
                                                  in-bounds?
                                                  burnable-neighbors?
                                                  get-neighbors
                                                  get-value-at
                                                  sample-at
                                                  distance-3d]]
            [gridfire.crown-fire          :refer [crown-fire-eccentricity
                                                  crown-fire-line-intensity
                                                  cruz-crown-fire-spread
                                                  van-wagner-crown-fire-initiation?]]
            [gridfire.fuel-models         :refer [build-fuel-model moisturize]]
            [gridfire.perturbation        :as perturbation]
            [gridfire.spotting            :as spot]
            [gridfire.surface-fire        :refer [anderson-flame-depth
                                                  byram-fire-line-intensity
                                                  byram-flame-length
                                                  rothermel-surface-fire-spread-any
                                                  rothermel-surface-fire-spread-max
                                                  rothermel-surface-fire-spread-no-wind-no-slope
                                                  wind-adjustment-factor]]
            [gridfire.utils.random        :as random])
  (:import java.util.Random))

(m/set-current-implementation :vectorz)

;; for surface fire, tau = 10 mins, t0 = 0, and t = global-clock
;; for crown fire, tau = 20 mins, t0 = time of first torch, t = global-clock
;; (defn lautenberger-spread-acceleration
;;   [equilibrium-spread-rate t0 t tau]
;;   (* equilibrium-spread-rate (- 1.0 (Math/exp (/ (- t0 t 0.2) tau)))))
;;
;; Note: Because of our use of adaptive timesteps, if the spread rate on
;;       the first timestep is not at least 83 ft/min, then the timestep will
;;       be calculated as greater than 60 minutes, which will terminate the
;;       one hour fire simulation instantly.

(defn random-cell
  "Returns a random [i j] pair with i < num-rows and j < num-cols."
  [num-rows num-cols]
  [(rand-int num-rows)
   (rand-int num-cols)])

(def offset-to-degrees
  "Returns clockwise degrees from north."
  {[-1  0]   0.0   ; N
   [-1  1]  45.0   ; NE
   [ 0  1]  90.0   ; E
   [ 1  1] 135.0   ; SE
   [ 1  0] 180.0   ; S
   [ 1 -1] 225.0   ; SW
   [ 0 -1] 270.0   ; W
   [-1 -1] 315.0}) ; NW

(defn rothermel-fast-wrapper
  [fuel-model-number fuel-moisture]
  (let [fuel-model      (-> (build-fuel-model (int fuel-model-number))
                            (moisturize fuel-moisture))
        spread-info-min (rothermel-surface-fire-spread-no-wind-no-slope fuel-model)]
    [fuel-model spread-info-min]))

(defrecord BurnTrajectory
    [cell
     source
     trajectory
     ^double terrain-distance
     ^double spread-rate
     ^double fire-line-intensity
     ^double flame-length
     fractional-distance
     fire-type
     crown-fire?])
```

```
(defn compute-burn-trajectory
  [neighbor here spread-info-min spread-info-max fuel-model crown-bulk-density
   canopy-cover canopy-height canopy-base-height foliar-moisture crown-spread-max
   crown-eccentricity landfire-rasters cell-size overflow-trajectory overflow-heat
   crown-type]
  (let [trajectory                (mapv - neighbor here)
        spread-direction          (offset-to-degrees trajectory)
        surface-spread-rate       (rothermel-surface-fire-spread-any spread-info-max
                                                                     spread-direction)
        residence-time            (:residence-time spread-info-min)
        reaction-intensity        (:reaction-intensity spread-info-min)
        surface-intensity         (->> (anderson-flame-depth surface-spread-rate residence-time)
                                       (byram-fire-line-intensity reaction-intensity))
        crown-fire?               (van-wagner-crown-fire-initiation? canopy-cover
                                                                     canopy-base-height
                                                                     foliar-moisture
                                                                     surface-intensity)
        ^double crown-spread-rate (when crown-fire?
                                    (rothermel-surface-fire-spread-any
                                     (assoc spread-info-max
                                            :max-spread-rate crown-spread-max
                                            :eccentricity crown-eccentricity)
                                     spread-direction))
        ^double crown-intensity   (when crown-fire?
                                    (crown-fire-line-intensity
                                     crown-spread-rate
                                     crown-bulk-density
                                     canopy-height
                                     canopy-base-height
                                     (-> fuel-model :h :dead :1hr)))
        spread-rate               (if crown-fire?
                                    (max surface-spread-rate crown-spread-rate)
                                    surface-spread-rate)
        fire-line-intensity       (if crown-fire?
                                    (+ surface-intensity crown-intensity)
                                    surface-intensity)
        flame-length              (byram-flame-length fire-line-intensity)]
    (->BurnTrajectory neighbor
                      here
                      trajectory
                      (distance-3d (:elevation landfire-rasters) cell-size here neighbor)
                      spread-rate
                      fire-line-intensity
                      flame-length
                      (volatile! (if (= trajectory overflow-trajectory)
                                   overflow-heat
                                   0.0))
                      (if crown-fire? crown-type :surface)
                      crown-fire?)))

(defn compute-neighborhood-fire-spread-rates!
  "Returns a vector of entries of the form:
  {:cell [i j],
   :trajectory [di dj],
   :terrain-distance ft,
   :spread-rate ft/min,
   :fire-line-intensity Btu/ft/s,
   :flame-length ft,
   :fractional-distance [0-1]}, one for each cell adjacent to here."
  [{:keys [landfire-rasters multiplier-lookup perturbations wind-speed-20ft wind-from-direction
           temperature relative-humidity foliar-moisture ellipse-adjustment-factor
           cell-size num-rows num-cols] :as constants}
   fire-spread-matrix
   here
   overflow-trajectory
   overflow-heat
   global-clock]
  (let [^double aspect                (sample-at here
                                                 global-clock
                                                 (:aspect landfire-rasters)
                                                 (:aspect multiplier-lookup)
                                                 (:aspect perturbations))
        ^double canopy-base-height    (sample-at here
                                                 global-clock
                                                 (:canopy-base-height landfire-rasters)
                                                 (:canopy-base-height multiplier-lookup)
                                                 (:canopy-base-height perturbations))
        ^double canopy-cover          (sample-at here
                                                 global-clock
                                                 (:canopy-cover landfire-rasters)
                                                 (:canopy-cover multiplier-lookup)
                                                 (:canopy-cover perturbations))
        ^double canopy-height         (sample-at here
                                                 global-clock
                                                 (:canopy-height landfire-rasters)
                                                 (:canopy-height multiplier-lookup)
                                                 (:canopy-height perturbations))
        ^double crown-bulk-density    (sample-at here
                                                 global-clock
                                                 (:crown-bulk-density landfire-rasters)
                                                 (:crown-bulk-density multiplier-lookup)
                                                 (:crown-bulk-density perturbations))
        ^long fuel-model              (sample-at here
                                                 global-clock
                                                 (:fuel-model landfire-rasters)
                                                 (:fuel-model multiplier-lookup)
                                                 (:fuel-model perturbations))
        ^double slope                 (sample-at here
                                                 global-clock
                                                 (:slope landfire-rasters)
                                                 (:slope multiplier-lookup)
                                                 (:slope perturbations))
        ^double relative-humidity     (get-value-at here
                                                    global-clock
                                                    relative-humidity
                                                    (:relative-humidity multiplier-lookup)
                                                    (:relative-humidity perturbations))
        ^double temperature           (get-value-at here
                                                    global-clock
                                                    temperature
                                                    (:temperature multiplier-lookup)
                                                    (:temperature perturbations))
        ^double wind-from-direction   (get-value-at here
                                                    global-clock
                                                    wind-from-direction
                                                    (:wind-from-direction multiplier-lookup)
                                                    (:wind-from-direction perturbations))
        ^double wind-speed-20ft       (get-value-at here
                                                    global-clock
                                                    wind-speed-20ft
                                                    (:wind-speed-20ft multiplier-lookup)
                                                    (:wind-speed-20ft perturbations))
        ^double fuel-moisture         (or (fuel-moisture-from-raster constants here global-clock)
                                          (get-fuel-moisture relative-humidity temperature))
        [fuel-model spread-info-min]  (rothermel-fast-wrapper fuel-model fuel-moisture)
        midflame-wind-speed           (* wind-speed-20ft
                                         88.0
                                         (wind-adjustment-factor ^long (:delta fuel-model)
                                                                 canopy-height canopy-cover)) ; mi/hr -> ft/min
        spread-info-max               (rothermel-surface-fire-spread-max spread-info-min
                                                                         midflame-wind-speed
                                                                         wind-from-direction
                                                                         slope
                                                                         aspect
                                                                         ellipse-adjustment-factor)
        [crown-type crown-spread-max] (cruz-crown-fire-spread wind-speed-20ft crown-bulk-density
                                                          (-> fuel-moisture :dead :1hr))
        crown-eccentricity            (crown-fire-eccentricity wind-speed-20ft
                                                           ellipse-adjustment-factor)]
    (into []
          (comp
           (filter #(and (in-bounds? num-rows num-cols %)
                         (burnable? fire-spread-matrix (:fuel-model landfire-rasters) here %)))
           (map #(compute-burn-trajectory % here spread-info-min spread-info-max fuel-model
                                          crown-bulk-density canopy-cover canopy-height
                                          canopy-base-height foliar-moisture crown-spread-max
                                          crown-eccentricity landfire-rasters cell-size
                                          overflow-trajectory overflow-heat crown-type)))
          (get-neighbors here))))

(defn- get-old-fractional-distance
  [{:keys [trajectory-combination]} {:keys [fractional-distance]} fractional-distance-matrix [i j]]
  (if (= trajectory-combination :sum)
    (m/mget fractional-distance-matrix i j)
    @fractional-distance))

(defn- update-fractional-distance-matrix!
  "Update the fractional distance matrix with the largest fractional distance calculated."
  [fractional-distance-matrix max-fractionals]
  (doseq [[cell fractional-distance] @max-fractionals]
    (let [[i j] cell]
      (m/mset! fractional-distance-matrix i j fractional-distance))))

(defn- update-fractional-distance!
  "Update fractional distance for given trajectory into the current cell. Return a tuple of [old-value new-value]"
  [{:keys [trajectory-combination] :as inputs} max-fractionals trajectory fractional-distance-matrix timestep cell]
  (let [terrain-distance    (double (:terrain-distance trajectory))
        spread-rate         (double (:spread-rate trajectory))
        new-spread-fraction (/ (* spread-rate timestep) terrain-distance)
        old-total           (get-old-fractional-distance inputs trajectory fractional-distance-matrix cell)
        new-total           (+ old-total new-spread-fraction)]
    (if (= trajectory-combination :sum)
      (let [max-fractional-distance  (max (get @max-fractionals cell 0.0) new-total)]
        (swap! max-fractionals assoc cell max-fractional-distance))
      (vreset! (:fractional-distance trajectory) new-total))
    [old-total new-total]))

(defn- update-overflow-heat
  [{:keys [num-rows num-cols]} fractional-distance-matrix {:keys [cell trajectory]} fractional-distance]
  (let [[i j :as target] (mapv + cell trajectory)]
    (when (in-bounds? num-rows num-cols target)
     (m/mset! fractional-distance-matrix i j (- fractional-distance 1.0)))))

(defn ignition-event-reducer
  [inputs max-fractionals fractional-distance-matrix timestep trajectory-combination fire-spread-matrix
   acc trajectory]
  (let [{:keys [source cell]} trajectory
        [i j]                 source
        [^double old-total ^double new-total] (update-fractional-distance! inputs
                                                                           max-fractionals
                                                                           trajectory
                                                                           fractional-distance-matrix
                                                                           timestep
                                                                           cell)]
    (if (and (>= new-total 1.0)
             (> new-total ^double (get-in acc [cell :fractional-distance] 0.0)))
      (do (when (and (= trajectory-combination :sum) (> new-total 1.0))
            (update-overflow-heat inputs fractional-distance-matrix trajectory new-total))
          (assoc! acc cell (merge trajectory {:fractional-distance  new-total
                                              :dt-adjusted          (* (/ (- 1.0 old-total) (- new-total old-total))
                                                                       timestep)
                                              :ignition-probability (m/mget fire-spread-matrix i j)})))
      acc)))
```

```
(defn identify-ignition-events
  [{:keys [trajectory-combination] :as inputs} ignited-cells timestep fire-spread-matrix fractional-distance-matrix]
  (let [timestep        (double timestep)
        max-fractionals (atom {})
        reducer-fn      (fn [acc trajectory]
                          (ignition-event-reducer inputs max-fractionals fractional-distance-matrix
                                                  timestep trajectory-combination fire-spread-matrix
                                                  acc trajectory))
        ignition-events (->> ignited-cells
                             (reduce reducer-fn (transient {}))
                             persistent!
                             vals)]
    (when (= trajectory-combination :sum)
      (update-fractional-distance-matrix! fractional-distance-matrix max-fractionals))
    ignition-events))

(defn update-ignited-cells
  [{:keys [landfire-rasters num-rows num-cols parallel-strategy] :as constants}
   ignited-cells
   ignition-events
   fire-spread-matrix
   global-clock]
  (let [fuel-model-matrix    (:fuel-model landfire-rasters)
        parallel-bin-size    (max 1 (quot (count ignition-events) (.availableProcessors (Runtime/getRuntime))))
        newly-ignited-cells  (into #{} (map :cell) ignition-events)
        pruned-ignited-cells (into [] (remove #(contains? newly-ignited-cells (:cell %))) ignited-cells)
        reducer-fn           (if (= parallel-strategy :within-fires)
                               #(->> (r/fold parallel-bin-size r/cat r/append! %)
                                     (reduce (fn [acc v] (into acc v)) pruned-ignited-cells))
                               #(reduce (fn [acc v] (into acc v)) pruned-ignited-cells %))]
    (->> ignition-events
         (r/map (fn [{:keys [cell trajectory fractional-distance]}]
                  (let [fractional-distance (double fractional-distance)]
                   (when (burnable-neighbors? fire-spread-matrix
                                              fuel-model-matrix
                                              num-rows num-cols
                                              cell)
                     (compute-neighborhood-fire-spread-rates!
                      constants
                      fire-spread-matrix
                      cell
                      trajectory
                      (- fractional-distance 1.0)
                      global-clock)))))
         (r/remove nil?)
         (reducer-fn))))

(defn generate-ignited-cells
  [constants fire-spread-matrix cells]
  (when (seq cells)
    (reduce (fn [ignited-cells cell]
              (into ignited-cells
                    (compute-neighborhood-fire-spread-rates! constants
                                                             fire-spread-matrix
                                                             cell
                                                             nil
                                                             0.0
                                                             0.0)))
            []
            cells)))

(defn identify-spot-ignition-events
  [global-clock spot-ignitions]
  (let [to-ignite-now (group-by (fn [[_ [time _]]]
                                  (let [time (double time)]
                                    (>= ^double global-clock time)))
                                spot-ignitions)
        ignite-later  (into {} (get to-ignite-now false))
        ignite-now    (into {} (get to-ignite-now true))]
    [ignite-later ignite-now]))

(defn spot-ignited-cells
  "Updates matrices for spot ignited cells
  Returns a map of ignited cells"
  [constants
   global-clock
   {:keys [fire-spread-matrix burn-time-matrix spread-rate-matrix fire-type-matrix
           flame-length-matrix fire-line-intensity-matrix spot-matrix]}
   spot-ignite-now]
  (let [ignited?        (fn [[k v]]
                          (let [[i j] k
                                [_ p] v]
                            (> ^double (m/mget fire-spread-matrix i j) ^double p)))
        spot-ignite-now (remove ignited? spot-ignite-now)
        ignited-cells   (generate-ignited-cells constants
                                                fire-spread-matrix
                                                (keys spot-ignite-now))]
    (doseq [cell spot-ignite-now
            :let [[i j]                    (key cell)
                  [_ ignition-probability] (val cell)]]
      (m/mset! fire-spread-matrix i j ignition-probability)
      (m/mset! burn-time-matrix i j global-clock)
      (m/mset! flame-length-matrix i j 1.0)
      (m/mset! fire-line-intensity-matrix i j 1.0)
      (m/mset! spread-rate-matrix i j -1.0)
      (m/mset! fire-type-matrix i j -1.0)
      (m/mset! spot-matrix i j 1.0))
    ignited-cells))

(defn new-spot-ignitions
  "Returns a map of [x y] locations to [t p] where:
  t: time of ignition
  p: ignition-probability"
  [{:keys [spotting] :as inputs} matrices ignition-events]
  (when spotting
    (reduce (fn [acc ignition-event]
              (merge-with (partial min-key first)
                          acc
                          (->> (spot/spread-firebrands
                                inputs
                                matrices
                                ignition-event)
                               (into {}))))
            {}
            ignition-events)))

(def fire-type-to-value
  {:surface       1.0
   :passive-crown 2.0
   :active-crown  3.0})

(defn- reducer-fn ^double
  [^double max-spread-rate ignited-cell]
  (Math/max max-spread-rate (double (:spread-rate ignited-cell))))

(defn run-loop
  [{:keys [max-runtime cell-size ignition-start-time] :as inputs}
   {:keys [fire-spread-matrix
           flame-length-matrix
           fire-line-intensity-matrix
           burn-time-matrix
           spread-rate-matrix
           fire-type-matrix
           fractional-distance-matrix
           spot-matrix] :as matrices}
   ignited-cells]
  (let [max-runtime        (double max-runtime)
        cell-size          (double cell-size)
        crown-fire-count   (atom 0)
        spot-count         (atom 0)
        ignition-stop-time (+ ignition-start-time max-runtime)]
    (loop [global-clock  ignition-start-time
           ignited-cells  ignited-cells
           spot-ignitions {}]
      (if (and (< global-clock ignition-stop-time)
               (seq ignited-cells))
        (let [dt                (->> ignited-cells
                                     (reduce reducer-fn 0.0)
                                     (/ cell-size)
                                     double)
              timestep          (min dt (- ignition-stop-time global-clock))
              next-global-clock (+ global-clock timestep)
              ignition-events   (identify-ignition-events inputs ignited-cells timestep
                                                          fire-spread-matrix fractional-distance-matrix)
              inputs            (perturbation/update-global-vals inputs global-clock next-global-clock)]
          ;; [{:cell :trajectory :fractional-distance
          ;;   :flame-length :fire-line-intensity} ...]
          (doseq [{:keys
                   [cell flame-length fire-line-intensity
                    ignition-probability spread-rate fire-type
                    dt-adjusted crown-fire?]} ignition-events]
            (let [[i j]       cell
                  dt-adjusted (double dt-adjusted)]
              (when crown-fire? (swap! crown-fire-count inc))
              (m/mset! fire-spread-matrix         i j ignition-probability)
              (m/mset! flame-length-matrix        i j flame-length)
              (m/mset! fire-line-intensity-matrix i j fire-line-intensity)
              (m/mset! burn-time-matrix           i j (+ global-clock dt-adjusted))
              (m/mset! spread-rate-matrix         i j spread-rate)
              (m/mset! fire-type-matrix           i j (fire-type fire-type-to-value))))
          (let [new-spot-ignitions (new-spot-ignitions (assoc inputs :global-clock global-clock)
                                                       matrices
                                                       ignition-events)
                [spot-ignite-later
                 spot-ignite-now]  (identify-spot-ignition-events global-clock
                                                                  (merge-with (partial min-key first)
                                                                              spot-ignitions
                                                                              new-spot-ignitions))
                spot-ignited-cells (spot-ignited-cells inputs
                                                       global-clock
                                                       matrices
                                                       spot-ignite-now)]
            (reset! spot-count (+ @spot-count (count spot-ignited-cells)))
            (recur next-global-clock
                   (update-ignited-cells inputs
                                         (into spot-ignited-cells ignited-cells)
                                         ignition-events
                                         fire-spread-matrix
                                         global-clock)
                   spot-ignite-later)))
        {:global-clock               global-clock
         :exit-condition             (if (seq ignited-cells) :max-runtime-reached :no-burnable-fuels)
         :fire-spread-matrix         fire-spread-matrix
         :flame-length-matrix        flame-length-matrix
         :fire-line-intensity-matrix fire-line-intensity-matrix
         :burn-time-matrix           burn-time-matrix
         :spot-matrix                spot-matrix
         :spread-rate-matrix         spread-rate-matrix
         :fire-type-matrix           fire-type-matrix
         :crown-fire-count           @crown-fire-count
         :spot-count                 @spot-count}))))

(defn- initialize-matrix
  [num-rows num-cols indices]
  (let [matrix (m/zero-matrix num-rows num-cols)]
    (doseq [[i j] indices
            :when (in-bounds? num-rows num-cols [i j])]
      (m/mset! matrix i j -1.0))
    matrix))

(defn- get-non-zero-indices [m]
  (for [[r cols] (map-indexed vector (m/non-zero-indices m))
        c        cols]
    [r c]))

(defmulti run-fire-spread
  "Runs the raster-based fire spread model with a map of these arguments:
  - max-runtime: double (minutes)
  - cell-size: double (feet)
  - landfire-rasters: map containing these entries;
    - elevation: core.matrix 2D double array (feet)
    - slope: core.matrix 2D double array (vertical feet/horizontal feet)
    - aspect: core.matrix 2D double array (degrees clockwise from north)
    - fuel-model: core.matrix 2D double array (fuel model numbers 1-256)
    - canopy-height: core.matrix 2D double array (feet)
    - canopy-base-height: core.matrix 2D double array (feet)
    - crown-bulk-density: core.matrix 2D double array (lb/ft^3)
    - canopy-cover: core.matrix 2D double array (0-100)
  - wind-speed-20ft: double (miles/hour)
  - wind-from-direction: double (degrees clockwise from north)
  - fuel-moisture: doubles (%){:dead {:1hr :10hr :100hr} :live {:herbaceous :woody}}
  - foliar-moisture: double (%)
  - ellipse-adjustment-factor: (< 1.0 = more circular, > 1.0 = more elliptical)
  - initial-ignition-site: One of the following:
     - point represented as [row col]
     - map containing a :matrix field of type core.matrix 2D double array (0-2)
     - nil (this causes GridFire to select a random ignition-point)
  - num-rows: integer
  - num-cols: integer"
  (fn [{:keys [initial-ignition-site]}]
    (condp = (type initial-ignition-site)
      clojure.lang.PersistentHashMap :ignition-perimeter
      clojure.lang.PersistentVector  :ignition-point
      :random-ignition-point)))

(defmethod run-fire-spread :random-ignition-point
  [{:keys [ignitable-sites ^Random rand-gen] :as inputs}]
  (.nextDouble rand-gen)
  (run-fire-spread (assoc inputs
                          :initial-ignition-site
                          (random/my-rand-nth rand-gen ignitable-sites))))

(defmethod run-fire-spread :ignition-point
  [{:keys [landfire-rasters num-rows num-cols initial-ignition-site spotting trajectory-combination] :as inputs}]
  (let [[i j]                       initial-ignition-site
        fuel-model-matrix           (:fuel-model landfire-rasters)
        fire-spread-matrix          (m/zero-matrix num-rows num-cols)
        flame-length-matrix         (m/zero-matrix num-rows num-cols)
        fire-line-intensity-matrix (m/zero-matrix num-rows num-cols)
        burn-time-matrix            (m/zero-matrix num-rows num-cols)
        firebrand-count-matrix      (when spotting (m/zero-matrix num-rows num-cols))
        spread-rate-matrix          (m/zero-matrix num-rows num-cols)
        fire-type-matrix            (m/zero-matrix num-rows num-cols)
        spot-matrix                 (m/zero-matrix num-rows num-cols)
        fractional-distance-matrix (when (= trajectory-combination :sum) (m/zero-matrix num-rows num-cols))]
    (when (and (in-bounds? num-rows num-cols initial-ignition-site)
               (burnable-fuel-model? (m/mget fuel-model-matrix i j))
               (burnable-neighbors? fire-spread-matrix fuel-model-matrix
                                    num-rows num-cols initial-ignition-site))
      ;; initialize the ignition site
      (m/mset! fire-spread-matrix i j 1.0)
      (m/mset! flame-length-matrix i j 1.0)
      (m/mset! fire-line-intensity-matrix i j 1.0)
      (m/mset! burn-time-matrix i j -1.0)
      (m/mset! spread-rate-matrix i j -1.0)
      (m/mset! fire-type-matrix i j -1.0)
      (let [ignited-cells (compute-neighborhood-fire-spread-rates!
                           inputs
                           fire-spread-matrix
                           initial-ignition-site
                           nil
                           0.0
                           0.0)]
        (run-loop inputs
                  {:fire-spread-matrix         fire-spread-matrix
                   :spread-rate-matrix         spread-rate-matrix
                   :flame-length-matrix        flame-length-matrix
                   :fire-line-intensity-matrix fire-line-intensity-matrix
                   :firebrand-count-matrix     firebrand-count-matrix
                   :burn-time-matrix           burn-time-matrix
                   :fire-type-matrix           fire-type-matrix
                   :fractional-distance-matrix fractional-distance-matrix
                   :spot-matrix                spot-matrix}
                  ignited-cells)))))

(defmethod run-fire-spread :ignition-perimeter
  [{:keys [num-rows num-cols initial-ignition-site landfire-rasters spotting trajectory-combination] :as inputs}]
  (let [fire-spread-matrix         (first (m/mutable (:matrix initial-ignition-site)))
        non-zero-indices           (get-non-zero-indices fire-spread-matrix)
        perimeter-indices          (filter #(burnable-neighbors? fire-spread-matrix
                                                                 (:fuel-model landfire-rasters)
                                                                 num-rows
                                                                 num-cols
                                                                 %)
                                           non-zero-indices)]
    (when (seq perimeter-indices)
      (let [flame-length-matrix        (initialize-matrix num-rows num-cols non-zero-indices)
            fire-line-intensity-matrix (initialize-matrix num-rows num-cols non-zero-indices)
            burn-time-matrix           (initialize-matrix num-rows num-cols non-zero-indices)
            firebrand-count-matrix     (when spotting (m/zero-matrix num-rows num-cols))
            spread-rate-matrix         (initialize-matrix num-rows num-cols non-zero-indices)
            fire-type-matrix           (initialize-matrix num-rows num-cols non-zero-indices)
            fractional-distance-matrix (when (= trajectory-combination :sum)
                                         (initialize-matrix num-rows num-cols non-zero-indices))
            spot-matrix                (m/zero-matrix num-rows num-cols)
            ignited-cells              (generate-ignited-cells inputs fire-spread-matrix perimeter-indices)]
        (when (seq ignited-cells)
          (run-loop inputs
                    {:fire-spread-matrix         fire-spread-matrix
                     :spread-rate-matrix         spread-rate-matrix
                     :flame-length-matrix        flame-length-matrix
                     :fire-line-intensity-matrix fire-line-intensity-matrix
                     :firebrand-count-matrix     firebrand-count-matrix
                     :burn-time-matrix           burn-time-matrix
                     :fire-type-matrix           fire-type-matrix
                     :fractional-distance-matrix fractional-distance-matrix
                     :spot-matrix                spot-matrix}
                    ignited-cells))))))
```

This concludes our description of GridFire's raster-based fire spread algorithm.

5.5 Spotting Model Forumulas

Gridfire can optionally include spot fires using a cellular automata model described in Perryman 2013. The model is broken up into four submodels: Surface Spread, Tree Torching, Firebrand Dispersal, and Spot Ignition. For Surface Spread and Tree Torching, the Perryman model uses Rothermal (1972) and Van Wagner 1977 respectively. Gridfire will use the same models described in the previous sections.

The Firebrand Dispersal model describes the distributions of firebrands relative to the wind direction. The location of where the firebrand lands is determined by the probabilties of landing d meters in the direction parallel and perpendicular to the wind.

For determining the distance a firebrands should land parallel to the wind a lognormal probability density function is used from Sardoy (2008). Instead of calculating the probability GridFire will sample using a log-normal distribution using the mean and standard deviations derived from the fireline intensity and wind speed (Sardoy 2008).

Mean and spotting distance (m) and it's variance (v):

$$m = aQ^b * U^c$$

$$v = m * d$$

The emperical parameters a,b,c, and d is specified directly (see section 8 in Confguration File)

a = mean-distance

b = flin-exp

c = ws-exp

d = normalized-distance-variance

The normalized mean (μ) and standard deviation (σ) of the lognormal distribution are then calculated from m and v as:

$$\mu = ln(\frac{m^2}{\sqrt{v+m^2}})$$

$$\sigma = \sqrt{ln(1+\frac{v}{m^2})}$$

The above values are used to plugged into the lognormal distribution function:

$$f(d) = \frac{1}{\sqrt{2\pi\sigma x}} exp(-\frac{1}{2}\frac{ln(d)-\mu}{\sigma}^2)$$

Instead of implementing this function Gridfire uses the log-normal function from kixi.stats (a Clojure/Clojurescript library of statistical sampling and transducing functions).

For determining the distance a firebrands should land perpendicular to the wind a normal distribution with the mean of 0 and standard deviation of 0.92 is used, as described in Himoto and Tanaka (2005) (referenced in Perryman).

Once we have the mean and standard deviation we can sample using log-normal distribution for the direction parallel to the wind and normal distribution for the direction perpendicular to the wind.

```
(ns gridfire.spotting
  (:require [clojure.core.matrix :as m]
            [gridfire.common :refer [distance-3d
                                     get-fuel-moisture
                                     get-value-at
                                     in-bounds?
                                     burnable?
                                     fuel-moisture-from-raster]]
            [gridfire.utils.random :refer [random-float my-rand-range]]
            [gridfire.conversion :as convert]
            [kixi.stats.distribution :as distribution]))

(m/set-current-implementation :vectorz)

;;-----------------------------------------------------------------------------
;; Formulas
;;-----------------------------------------------------------------------------

(defn- sample-spotting-params
  ^double
  [param rand-gen]
  (if (map? param)
    (let [{:keys [lo hi]} param
          l                (if (vector? lo) (my-rand-range rand-gen lo) lo)
          h                (if (vector? hi) (my-rand-range rand-gen hi) hi)]
      (my-rand-range rand-gen [l h]))
    param))

(defn- mean-variance
  "Returns mean spotting distance and it's variance given:
  fire-line-intensity: (kWm^-1)
  wind-speed-20ft: (ms^-1)"
  [{:keys [^double mean-distance ^double flin-exp ^double ws-exp ^double normalized-distance-variance]}
   rand-gen ^double fire-line-intensity ^double wind-speed-20ft]
  (let [a (sample-spotting-params mean-distance rand-gen)
        b (sample-spotting-params flin-exp rand-gen)
        c (sample-spotting-params ws-exp rand-gen)
        m (* a (Math/pow fire-line-intensity b) (Math/pow wind-speed-20ft c))]
    {:mean m :variance (* m (sample-spotting-params normalized-distance-variance rand-gen))}))

(defn- standard-deviation
  "Returns standard deviation for the lognormal distribution given:
  mean spotting distance and it's variance"
  ^double
  [^double m ^double v]
  (Math/sqrt (Math/log (+ 1 (/ v (Math/pow m 2))))))

(defn- normalized-mean
  "Returns normalized mean for the lognormal distribution given:
  mean spotting distance and it's variance"
  ^double
  [^double m ^double v]
  (Math/log (/ (Math/pow m 2)
               (Math/sqrt (+ v (Math/pow m 2))))))

(defn- sample-wind-dir-deltas
  "Returns a sequence of [x y] distances (meters) that firebrands land away
  from a torched cell at i j where:
  x: parallel to the wind
  y: perpendicular to the wind (positive values are to the right of wind direction)"
  [{:keys [spotting rand-gen random-seed]}
   fire-line-intensity-matrix
   wind-speed-20ft [i j]]
  (let [num-firebrands          (sample-spotting-params (:num-firebrands spotting) rand-gen)
        intensity               (convert/Btu-ft-s->kW-m (m/mget fire-line-intensity-matrix i j))
        {:keys [mean variance]} (mean-variance spotting rand-gen intensity wind-speed-20ft)
        mu                      (normalized-mean mean variance)
        sd                      (standard-deviation mean variance)
        parallel                (distribution/log-normal {:mu mu :sd sd})
        perpendicular           (distribution/normal {:mu 0 :sd 0.92})
        parallel-values         (distribution/sample num-firebrands parallel {:seed random-seed})
        perpendicular-values    (distribution/sample num-firebrands perpendicular {:seed random-seed})]
    (mapv (fn [x y] [(convert/m->ft x) (convert/m->ft y)])
          parallel-values
          perpendicular-values)))
```

Since the results are distance deltas relative to the wind direction we must convert this to deltas in our coordinate plane. We can convert these deltas by using trigonometric functions.

```
(defn hypotenuse ^double
  [x y]
  (Math/sqrt (+ (Math/pow x 2) (Math/pow y 2))))

(defn deltas-wind->coord
  "Converts deltas from the torched tree in the wind direction to deltas
  in the coordinate plane"
  [deltas ^double wind-direction]
  (mapv (fn [[d-paral d-perp]]
          (let [d-paral (double d-paral)
                d-perp  (double d-perp)
                H  (hypotenuse d-paral d-perp)
                t1 wind-direction
                t2 (convert/rad->deg (Math/atan (/ d-perp d-paral)))
                t3 (+ t1 t2)]
           [(* -1 H (Math/cos (convert/deg->rad t3)))
            (* H (Math/sin (convert/deg->rad t3)))]))
        deltas))

(defn- firebrands
  "Returns a sequence of cells that firebrands land in"
  [deltas wind-towards-direction cell ^double cell-size]
  (let [step         (/ cell-size 2)
        [x y]        (mapv #(+ step (* ^double % step)) cell)
        x            (double x)
        y            (double y)
        coord-deltas (deltas-wind->coord deltas wind-towards-direction)]
    (mapv (fn [[dx dy]]
            (let [dx (double dx)
                  dy (double dy)]
              [(int (Math/floor (/ (+ dx x) step)))
               (int (Math/floor (/ (+ dy y) step)))]))
          coord-deltas)))
```

The Spot Ignition model describes the probability of a spot ignition as well as when the spot ignition should occur. Perryman uses the method described in Schroeder (1969) but adjusts the result to take into account the distance a firebrand lands from the source tree (using Albini 1979) and the number of firebrands that land in a cell (using Stauffer 2008).

$$P(I)_d = P(I)exp(-\lambda_s d)P(I)_d^{FB} = 1 - (1 - P(I)_d)^b$$

where λ is a positive number representing the decay constant, d is the firebrand's landing distance away from the source cell. $P(I)_d$ is the probability of spot ignition taking into consideration of d. $P(I)_d^{FB}$ is the probability of spot fire ignition taking into consideration b, the number of firebrands landing in a cell.

```
(defn- specific-heat-dry-fuel
  "Returns specific heat of dry fuel given:
  initiial-temp: (Celcius)
  ignition-temp: (Celcius)"
  ^double
  [^double initial-temp ^double ignition-temp]
  (+ 0.266 (* 0.0016 (/ (+ ignition-temp initial-temp) 2))))

(defn- heat-of-preignition
  "Returns heat of preignition given:
  init-temperature: (Celcius)
  ignition-temperature: (Celcius)
  moisture content: (Percent)"
  ^double
  [^double init-temperature ^double ignition-temperature ^double moisture]
  (let [T_o init-temperature
        T_i ignition-temperature
        M   moisture
        c_f (specific-heat-dry-fuel T_o T_i)

        ;; heat required to reach ignition temperature
        Q_a (* (- T_i T_o) c_f)

        ;; heat required to raise moisture to reach boiling point
        Q_b (* (- 100 T_o) M)

        ;; Heat of desorption
        Q_c (* 18.54 (- 1 (Math/exp (* -15.1 M))))

        ;; Heat required to vaporize moisture
        Q_d (* 540 M)]
    (+ Q_a Q_b Q_c Q_d)))

(defn- schroeder-ign-prob
  "Returns the probability of ignition as described in Shroeder (1969) given:
  relative-humidity: (%)
  temperature: (Farenheit)"
  ^double
  [fuel-moisture temperature]
  (let [ignition-temperature 320 ;;FIXME should this be a constant?
        moisture             (-> fuel-moisture :dead :1hr)
        Q_ig                 (heat-of-preignition (convert/F->C temperature) ignition-temperature moisture)
        X                    (/ (- 400 Q_ig) 10)]
    (/ (* 0.000048 (Math/pow X 4.3)) 50)))

(defn- spot-ignition-probability
  [{:keys [cell-size landfire-rasters]}
   {:keys [decay-constant]}
   fuel-moisture
   temperature
   firebrand-count
   torched-origin
   here]
  (let [ignition-probability (schroeder-ign-prob fuel-moisture temperature)
        distance             (convert/ft->m (distance-3d (:elevation landfire-rasters)
                                                         cell-size
                                                         here
                                                         torched-origin))
        decay-factor         (Math/exp (* -1 ^double decay-constant distance))]
    (- 1 (Math/pow (- 1 (* ignition-probability decay-factor)) firebrand-count))))
```

A firebrand will cause an unburened cell to transition to a burned state if the cell receives atleast one firebrand and the cell's probability of ignition as calculated by the above equations is greater than a randomly generated uniform number. Once a cell has been determined to ignite then the time until ignition is calculated. The time until ignition is a sum of three time intervals: the amount of time required for the firebrand to reach its maximum vertical height t_v, the amount of time required for the firebrand to descend from the maximum vertical height to the forest floor t_g, and the amount of time required for a spot fire to ignite and build up to the steady-state t_I. Perryman assumes t_v and t_g to be equal and used the formula from Albini (1979) to calculate it. t_I is also assumed to be 20 min as used in McAlpine and Wakimoto (1991).

```
(defn- spot-ignition?
  [rand-gen ^double spot-ignition-probability]
  (let [random-number (random-float 0 1 rand-gen)]
    (> spot-ignition-probability random-number)))

(defn- spot-ignition-time
  "Returns the time of spot ignition in minutes given:
  global-clock: (min)
  flame-length: (m)
  wind-speed-20ft: (ms^-1)"
  [^double global-clock ^double flame-length ^double wind-speed-20ft]
  (let [a              5.963
        b              (- a 1.4)
        D              0.003 ;firebrand diameter (m)
        z-max          (* 0.39 D (Math/pow 10 5))
        t-steady-state 20    ;period of building up to steady state from ignition (min)
        t_o            1     ;period of steady burning of tree crowns (min)
        t-max-height   (+ (/ t_o (/ (* 2 flame-length) wind-speed-20ft))
                          1.2
                          (* (/ a 3.0)
                             (- (Math/pow (/ (+ b (/ z-max flame-length)) a) (/ 3.0 2.0)) 1)))]
    (+ global-clock (* 2 t-max-height) t-steady-state)))
```

Once the locations, ignition probabilities, and time of ignition has been calculated for each of the firebrands a sequence of key value pairs are returned, to be processed in 'gridfire.cli'. The key is [x y] location of the firebrand and the value [t p] where t is the time of igintion and p is the ignition probability.

```
(defn- update-firebrand-counts!
  [{:keys [num-rows num-cols landfire-rasters]}
   firebrand-count-matrix
   fire-spread-matrix
   source
   firebrands]
  (doseq [[x y :as here] firebrands
          :when          (and (in-bounds? num-rows num-cols [x y])
                              (burnable? fire-spread-matrix
                                         (:fuel-model landfire-rasters)
                                         source
                                         here))
          :let           [new-count (inc ^double (m/mget firebrand-count-matrix x y))]]
    (m/mset! firebrand-count-matrix x y new-count)))

(defn- in-range?
  [[min max] fuel-model-number]
  (<= min fuel-model-number max))

(defn surface-spot-percent
  ^double
  [fuel-range-percents fuel-model-number rand-gen]
  (reduce (fn [acc [fuel-range percent]]
            (if (in-range? fuel-range fuel-model-number)
              (if (vector? percent)
                (my-rand-range rand-gen percent)
                percent)
              acc))
          0.0
          fuel-range-percents))

(defn- surface-fire-spot-fire?
  "Expects surface-fire-spotting config to be a sequence of tuples of
  ranges [lo hi] and spottting percent. The range represents the range (inclusive)
  of fuel model numbers that the spotting percent is set to.
  [[[1 140] 0.0]
  [[141 149] 1.0]
  [[150 256] 1.0]]"
  [{:keys [spotting rand-gen landfire-rasters]} [i j] ^double fire-line-intensity]
  (let [{:keys [surface-fire-spotting]} spotting]
    (when (and
           surface-fire-spotting
           (> fire-line-intensity ^double (:critical-fire-line-intensity surface-fire-spotting)))
      (let [fuel-range-percents (:spotting-percent surface-fire-spotting)
            fuel-model-raster   (:fuel-model landfire-rasters)
            fuel-model-number   (int (m/mget fuel-model-raster i j))
            spot-percent        (surface-spot-percent fuel-range-percents fuel-model-number rand-gen)]
        (>= spot-percent (random-float 0.0 1.0 rand-gen))))))

(defn- crown-spot-fire? [{:keys [spotting rand-gen]}]
  (when-let [spot-percent (:crown-fire-spotting-percent spotting)]
    (let [^double p (if (vector? spot-percent)
                      (let [[lo hi] spot-percent]
                        (random-float lo hi rand-gen))
                      spot-percent)]
      (>= p (random-float 0.0 1.0 rand-gen)))))

(defn- spot-fire? [inputs crown-fire? here fire-line-intensity]
  (if crown-fire?
    (crown-spot-fire? inputs)
    (surface-fire-spot-fire? inputs here fire-line-intensity)))

(defn spread-firebrands
  "Returns a sequence of key value pairs where
  key: [x y] locations of the cell
  val: [t p] where:
  t: time of ignition
  p: ignition-probability"
  [{:keys
    [num-rows num-cols cell-size landfire-rasters global-clock spotting rand-gen
     multiplier-lookup perturbations temperature relative-humidity wind-speed-20ft
     wind-from-direction] :as inputs}
   {:keys [firebrand-count-matrix fire-spread-matrix fire-line-intensity-matrix flame-length-matrix]}
   {:keys [cell fire-line-intensity crown-fire?]}]
  (when (spot-fire? inputs crown-fire? cell fire-line-intensity)
    (let [tmp               (get-value-at cell
                                          global-clock
                                          temperature
                                          (:temperature multiplier-lookup)
                                          (:temperature perturbations))
          rh                (get-value-at cell
                                          global-clock
                                          relative-humidity
                                          (:relative-humidity multiplier-lookup)
                                          (:relative-humidity perturbations))
          ws                (get-value-at cell
                                          global-clock
                                          wind-speed-20ft
                                          (:wind-speed-20ft multiplier-lookup)
                                          (:wind-speed-20ft perturbations))
          ^double
          wd                (get-value-at cell
                                          global-clock
                                          wind-from-direction
                                          (:wind-from-direction multiplier-lookup)
                                          (:wind-from-direction perturbations))
          fuel-moisture     (or (fuel-moisture-from-raster inputs cell global-clock)
                                (get-fuel-moisture rh temperature))
          deltas            (sample-wind-dir-deltas inputs
                                                    fire-line-intensity-matrix
                                                    (convert/mph->mps ws)
                                                    cell)
          wind-to-direction (mod (+ 180 wd) 360)
          firebrands        (firebrands deltas wind-to-direction cell cell-size)]
      (update-firebrand-counts! inputs firebrand-count-matrix fire-spread-matrix cell firebrands)
      (->> (for [[x y] firebrands
                 :when (and (in-bounds? num-rows num-cols [x y])
                            (burnable? fire-spread-matrix (:fuel-model landfire-rasters) cell [x y]))
                 :let  [firebrand-count (m/mget firebrand-count-matrix x y)
```

```
                        spot-ignition-p (spot-ignition-probability inputs
                                                                   spotting
                                                                   fuel-moisture
                                                                   tmp
                                                                   firebrand-count
                                                                   cell
                                                                   [x y])]]
              (when (spot-ignition? rand-gen spot-ignition-p)
                (let [[i j] cell
                      t     (spot-ignition-time global-clock
                                                (convert/ft->m (m/mget flame-length-matrix i j))
                                                (convert/mph->mps ws))]
                  [[x y] [t spot-ignition-p]]))))
            (remove nil?)))))
```

6 User Interface

The GridFire model described in the previous section may be called directly from the REPL through the **run-fire-spread** function. However, this would require that the user had already prepared all of their map layers as 2D Clojure core.matrix values. In order to enable GridFire to easily access a wide range of raster formatted GIS layers directly, we have the following options:

1. A simple Clojure interface to a Postgresql database, containing the PostGIS spatial extensions. This interface is described in Section 6.1.
2. Magellan, a Clojure library for interacting with geospatial datasets. This interface is described in Section 6.2.

Section 6.3 describes GridFire's command line interface along with its input configuration file format, which allows users to select between the PostGIS and Magellan data import options easily.

Using one of these options along with a simple client interface in clojure Section 6.3 which describes GridFire's command line interface along with its input configuration file format.

6.1 PostGIS Bridge

Extracting raster layers from a PostGIS database is performed by a single function, called **postgis-raster-to-matrix**, which constructs a SQL query for the layer, sends it to the database in a transaction, and returns the result as a core.matrix 2D double array with nodata values represented as -1.0. The georeferencing information associated with this tile is also included in the returned results. This function may be called directly from the REPL or indirectly through GridFire's command line interface.

```
(ns gridfire.postgis-bridge
  (:require [clojure.core.matrix :as m]
            [clojure.java.jdbc   :as jdbc])
  (:import org.postgresql.jdbc.PgArray
           java.util.UUID))

(m/set-current-implementation :vectorz)

(defn extract-matrix [result]
  (->> result
       :matrix
       (#(.getArray ^PgArray %))
       (m/emap #(or % -1.0))
       m/matrix))

(defn build-rescale-query [rescaled-table-name resolution table-name]
  (format (str "CREATE TEMPORARY TABLE %s "
               "ON COMMIT DROP AS "
               "SELECT ST_Rescale(rast,%s,-%s,'NearestNeighbor') AS rast "
               "FROM %s")
          rescaled-table-name
          resolution
          resolution
          table-name))

(defn build-threshold-query [threshold]
  (format (str "ST_MapAlgebra(rast,band,NULL,"
               "'CASE WHEN [rast.val] < %s"
               " THEN 0.0 ELSE [rast.val] END')")
          threshold))

(defn build-data-query [threshold threshold-query metadata table-name]
  (format (str "SELECT ST_DumpValues(%s,%s) AS matrix "
               "FROM generate_series(1,%s) AS band "
               "CROSS JOIN %s")
          (if threshold threshold-query "rast")
          (if threshold 1 "band")
          (:numbands metadata)
          table-name))

(defn build-meta-query [table-name]
  (format "SELECT (ST_Metadata(rast)).* FROM %s" table-name))

(defn postgis-raster-to-matrix
  "Send a SQL query to the PostGIS database given by db-spec for a
  raster tile from table table-name. Optionally resample the raster to
  match resolution and set any values below threshold to 0. Return the
  post-processed raster values as a Clojure matrix using the
  core.matrix API along with all of the georeferencing information
  associated with this tile in a hash-map with the following form:
  {:srid 900916,
   :upperleftx -321043.875,
   :upperlefty -1917341.5,
   :width 486,
   :height 534,
   :scalex 2000.0,
   :scaley -2000.0,
   :skewx 0.0,
   :skewy 0.0,
   :numbands 10,
   :matrix #vectorz/matrix Large matrix with shape: [10,534,486]}"
  [db-spec table-name & [resolution threshold]]
  (jdbc/with-db-transaction [conn db-spec]
    (let [table-name      (if-not resolution
                            table-name
                            (let [rescaled-table-name (str "gridfire_" (subs (str (UUID/randomUUID)) 0 8))
                                  rescale-query       (build-rescale-query rescaled-table-name resolution table-name)]
                              ;; Create a temporary table to hold the rescaled raster.
                              ;; It will be dropped when the transaction completes.
                              (jdbc/db-do-commands conn [rescale-query])
                              rescaled-table-name))
          meta-query      (build-meta-query table-name)
          metadata        (first (jdbc/query conn [meta-query]))
          threshold-query (build-threshold-query threshold)
          data-query      (build-data-query threshold threshold-query metadata table-name)
          matrix          (when-let [results (seq (jdbc/query conn [data-query]))]
                            (m/matrix (mapv extract-matrix results)))]
      (assoc metadata :matrix matrix))))
```

6.2 Magellan

Reading raster layers from disk is performed by a single function, called **geotiff-raster-to-matrix**. Given the location of a GeoTIFF file, this function will read the raster into memory and return the same map of information as the **postgis-raster-to-matrix** function, described in the previous section.

```
(ns gridfire.magellan-bridge
  (:require [clojure.core.matrix     :as m]
            [magellan.core           :refer [read-raster]]
            [magellan.raster.inspect :as inspect])
  (:import org.geotools.coverage.grid.GridGeometry2D
           org.geotools.referencing.operation.transform.AffineTransform2D))

(m/set-current-implementation :vectorz)

(defn geotiff-raster-to-matrix
  "Reads a raster from a file using the magellan.core library. Returns the
   post-processed raster values as a Clojure matrix using the core.matrix API
   along with all of the georeferencing information associated with this tile in a
   hash-map with the following form:
  {:srid 900916,
   :upperleftx -321043.875,
   :upperlefty -1917341.5,
   :width 486,
   :height 534,
   :scalex 2000.0,
   :scaley -2000.0,
   :skewx 0.0,
   :skewy 0.0,
   :numbands 10,
   :matrix #vectorz/matrix Large matrix with shape: [10,534,486]}"
  [file-path]
  (let [raster   (read-raster file-path)
        grid     ^GridGeometry2D (:grid raster)
        r-info   (inspect/describe-raster raster)
        matrix   (inspect/extract-matrix raster)
        image    (:image r-info)
        envelope (:envelope r-info)
        crs2d    ^AffineTransform2D (.getGridToCRS2D grid)]
    {:srid       (:srid r-info)
     :upperleftx (get-in envelope [:x :min])
     :upperlefty (get-in envelope [:y :max])
     :width      (:width image)
     :height     (:height image)
     :scalex     (.getScaleX crs2d)
     :scaley     (.getScaleY crs2d)
     :skewx      0.0                    ;FIXME not used?
     :skewy      0.0                    ;FIXME not used?
     :numbands   (:bands image)
     :matrix     (m/matrix matrix)}))
```

6.3 Command Line Interface

The entire GridFire system is available for use directly from the Clojure REPL. This enables straightforward analysis and introspection of the fire behavior functions and their results over a range of inputs. However, if you just want to simulate an individual ignition event, GridFire comes with a simple command line interface that can be parameterized by a single configuration file, specifying the ignition location, burn duration, weather values, and the location of the PostGIS raster layers to use for topography and fuels.

GridFire's command line interface can be built as an uberjar using the following command:

```
clojure -X:make-uberjar
```

The advantage of the uberjar format is that the single uberjar file can be shared easily between computers and can be run by anyone with a recent version of Java installed, without needing to install Clojure, Git, or any of the dependency libraries that GridFire uses.

The command above will output the uberjar into this repository's top level "target" directory. It can be run from the command line as follows:

```
java -jar gridfire.jar myconfig.edn
```

When run, the executable connects to the PostGIS database specified in the passed-in config file, downloads the necessary raster layers, simulates the ignition event for the requested duration, and returns 2D maps showing the spatial distributions of fire spread, flame length, and fire line intensity respectively. Finally, it prints out the final clock time from when the simulation was terminated as well as the total number of ignited cells on the raster grid at that point.

Which maps are created (and in what formats) may be configured by setting the following options in GridFire's input config file to true or false:

1. :output-landfire-inputs?
2. :output-geotiffs?
3. :output-pngs?

```
(ns gridfire.core
  (:require [clojure.core.matrix      :as m]
            [clojure.core.reducers    :as r]
            [clojure.data.csv         :as csv]
            [clojure.edn              :as edn]
            [clojure.java.io          :as io]
            [clojure.spec.alpha       :as spec]
            [clojure.string           :as str]
            [gridfire.binary-output   :as binary]
            [gridfire.common          :refer [calc-emc get-neighbors in-bounds?]]
            [gridfire.conversion      :refer [m->ft]]
            [gridfire.fetch           :as fetch]
            [gridfire.fire-spread     :refer [rothermel-fast-wrapper run-fire-spread]]
            [gridfire.perturbation    :as perturbation]
            [gridfire.random-ignition :as random-ignition]
            [gridfire.spec.config     :as config-spec]
            [gridfire.utils.random    :refer [draw-samples]]
            [magellan.core            :refer [make-envelope
                                              matrix-to-raster
                                              register-new-crs-definitions-from-properties-file!
                                              write-raster]]
            [matrix-viz.core          :refer [save-matrix-as-png]]
            [taoensso.tufte           :as tufte]
            [triangulum.logging       :refer [log log-str]])
  (:import java.util.Random))

(m/set-current-implementation :vectorz)

(register-new-crs-definitions-from-properties-file! "CUSTOM"
                                                    (io/resource "custom_projections.properties"))

(defn cells-to-acres
  [cell-size num-cells]
  (let [acres-per-cell (/ (* cell-size cell-size) 43560.0)]
    (* acres-per-cell num-cells)))

(defn summarize-fire-spread-results
  [fire-spread-results cell-size]
  (let [flame-lengths              (filterv pos? (m/eseq (:flame-length-matrix fire-spread-results)))
        fire-line-intensities      (filterv pos? (m/eseq (:fire-line-intensity-matrix fire-spread-results)))
        burned-cells               (count flame-lengths)
        fire-size                  (cells-to-acres cell-size burned-cells)
        crown-fire-size            (cells-to-acres cell-size (:crown-fire-count fire-spread-results))
        flame-length-mean          (/ (m/esum flame-lengths) burned-cells)
        fire-line-intensity-mean   (/ (m/esum fire-line-intensities) burned-cells)
        flame-length-stddev        (->> flame-lengths
                                        (m/emap #(Math/pow (- flame-length-mean %) 2.0))
                                        (m/esum)
                                        (#(/ % burned-cells))
                                        (Math/sqrt))
        fire-line-intensity-stddev (->> fire-line-intensities
                                        (m/emap #(Math/pow (- fire-line-intensity-mean %) 2.0))
                                        (m/esum)
                                        (#(/ % burned-cells))
                                        (Math/sqrt))]
    {:fire-size                  fire-size
     :crown-fire-size            crown-fire-size
     :flame-length-mean          flame-length-mean
     :flame-length-stddev        flame-length-stddev
     :fire-line-intensity-mean   fire-line-intensity-mean
     :fire-line-intensity-stddev fire-line-intensity-stddev
     :spot-count                 (:spot-count fire-spread-results)}))

(defn calc-ffwi
  "Computes the Fosberg Fire Weather Index value from rh (relative
   humidity in %), temp (temperature in F), wsp (wind speed in mph),
   and a constant x (gust multiplier).
   ------------------------------------------------------------------
   Note: ffwi can be computed with (calc-ffwi rh temp wsp 1.0)
         ffwi-max can be computed with (calc-ffwi minrh maxtemp wsp 1.75)
   Geek points: Uses Cramer's rule: (+ d (* x (+ c (* x (+ b (* x a))))))
                for an efficient cubic calculation on tmp."
  [rh temp wsp x]
  (let [m   (calc-emc rh temp)
        eta (+ 1 (* m (+ -2 (* m (+ 1.5 (* m -0.5))))))]
    (/ (* eta (Math/sqrt (+ 1 (Math/pow (* x wsp) 2))))
       0.3002)))

(defn kebab->snake [s]
  (str/replace s #"-" "_"))

(defn snake->kebab [s]
  (str/replace s #"_" "-"))

(defn min->hour [t]
  (int (quot t 60)))

(defn previous-active-perimeter?
  [[i j :as here] matrix]
  (let [num-rows (m/row-count matrix)
        num-cols (m/column-count matrix)]
    (and
     (= (m/mget matrix i j) -1.0)
     (->> (get-neighbors here)
          (filter #(in-bounds? num-rows num-cols %))
          (map #(apply m/mget matrix %))
          (some pos?)))))

(defn to-color-map-values [burn-time-matrix current-clock]
  (m/emap-indexed (fn [here burn-time]
                    (let [delta-hours (->> (- current-clock burn-time)
                                           min->hour)]
                      (cond
                        (previous-active-perimeter? here burn-time-matrix) 201
                        (= burn-time -1.0)                                 200
                        (< 0 delta-hours 5)                                delta-hours
                        (>= delta-hours 5)                                 5
                        :else                                              0)))
                  burn-time-matrix))
```

```
(defn layer-snapshot [burn-time-matrix layer-matrix t]
  (m/emap (fn [layer-value burn-time]
            (if (<= burn-time t)
              layer-value
              0))
          layer-matrix
          burn-time-matrix))

(defn output-filename [name outfile-suffix simulation-id output-time ext]
  (as-> [name outfile-suffix simulation-id (when output-time (str "t" output-time))] $
    (remove str/blank? $)
    (str/join "_" $)
    (str $ ext)))

(defn output-geotiff
  ([config matrix name envelope]
   (output-geotiff config matrix name envelope nil nil))

  ([config matrix name envelope simulation-id]
   (output-geotiff config matrix name envelope simulation-id nil))

  ([{:keys [output-directory outfile-suffix] :as config}
    matrix name envelope simulation-id output-time]
   (let [file-name (output-filename name
                                    outfile-suffix
                                    (str simulation-id)
                                    output-time
                                    ".tif")]
     (-> (matrix-to-raster name matrix envelope)
         (write-raster (if output-directory
                         (str/join "/" [output-directory file-name])
                         file-name))))))

(defn output-png
  ([config matrix name envelope]
   (output-png config matrix name envelope nil nil))

  ([config matrix name envelope simulation-id]
   (output-png config matrix name envelope simulation-id nil))

  ([{:keys [output-directory outfile-suffix]}
    matrix name envelope simulation-id output-time]
   (let [file-name (output-filename name
                                    outfile-suffix
                                    (str simulation-id)
                                    output-time
                                    ".png")]
     (save-matrix-as-png :color 4 -1.0
                         matrix
                         (if output-directory
                           (str/join "/" [output-directory file-name])
                           (file-name))))))

(def layer-name->matrix
  [["fire_spread"         :fire-spread-matrix]
   ["flame_length"        :flame-length-matrix]
   ["fire_line_intensity" :fire-line-intensity-matrix]
   ["burn_history"        :burn-time-matrix]
   ["spread_rate"         :spread-rate-matrix]
   ["fire_type"           :fire-type-matrix]])

(defn filter-output-layers [output-layers]
  (let [layers-to-filter (set (map (comp kebab->snake name) (keys output-layers)))]
    (filter (fn [[name _]] (contains? layers-to-filter name)) layer-name->matrix)))

(defn process-output-layers-timestepped
  [{:keys [simulation-id] :as config}
```

```
   {:keys [global-clock burn-time-matrix] :as fire-spread-results}
   name layer timestep envelope]
  (doseq [output-time (range 0 (inc global-clock) timestep)]
    (let [matrix          (if (= layer "burn_history")
                            (to-color-map-values layer output-time)
                            (fire-spread-results layer))
          filtered-matrix (layer-snapshot burn-time-matrix matrix output-time)]
      (output-geotiff config filtered-matrix name envelope simulation-id output-time)
      (output-png config filtered-matrix name envelope simulation-id output-time))))

(defn process-output-layers!
  [{:keys [output-layers output-geotiffs? output-pngs?] :as config}
   {:keys [global-clock burn-time-matrix] :as fire-spread-results}
   envelope
   simulation-id]
  (let [layers (if output-layers
                 (filter-output-layers output-layers)
                 layer-name->matrix)]
    (doseq [[name layer] layers]
      (let [kw       (keyword (snake->kebab name))
            timestep (get output-layers kw)]
        (if (int? timestep)
          (process-output-layers-timestepped config
                                             fire-spread-results
                                             name
                                             layer
                                             timestep
                                             envelope)
          (let [matrix (if (= layer "burn_history")
                         (to-color-map-values layer global-clock)
                         (fire-spread-results layer))]
            (when output-geotiffs?
             (output-geotiff config matrix name envelope simulation-id))
            (when output-pngs?
             (output-png config matrix name envelope simulation-id))))))))

(defn process-burn-count!
  [{:keys [fire-spread-matrix burn-time-matrix global-clock]}
   burn-count-matrix
   timestep]
  (if (int? timestep)
    (doseq [clock (range 0 (inc global-clock) timestep)]
      (let [filtered-fire-spread (m/emap (fn [layer-value burn-time]
                                           (if (<= burn-time clock)
                                             layer-value
                                             0))
                                         fire-spread-matrix
                                         burn-time-matrix)
            band                 (int (quot clock timestep))]
        (m/add! (nth (seq burn-count-matrix) band) filtered-fire-spread)))
    (m/add! burn-count-matrix fire-spread-matrix)))

(defn process-aggregate-output-layers!
  [{:keys [burn-count-matrix flame-length-max-matrix flame-length-sum-matrix
           output-burn-probability spot-count-matrix]} fire-spread-results]
  (when-let [timestep output-burn-probability]
    (process-burn-count! fire-spread-results burn-count-matrix timestep))
  (when flame-length-sum-matrix
    (m/add! flame-length-sum-matrix (:flame-length-matrix fire-spread-results)))
  (when flame-length-max-matrix
    (m/emap! #(max %1 %2) flame-length-max-matrix (:flame-length-matrix fire-spread-results)))
  (when spot-count-matrix
    (m/add! spot-count-matrix (:spot-matrix fire-spread-results))))

(defn initialize-burn-count-matrix
  [{:keys [output-burn-probability output-burn-count? max-runtimes num-rows num-cols]}]
  (when (or output-burn-count? output-burn-probability)
    (if (int? output-burn-probability)
      (let [num-bands (inc (quot (apply max max-runtimes) output-burn-probability))]
        (m/zero-array [num-bands num-rows num-cols]))
      (m/zero-array [num-rows num-cols]))))

(defn process-binary-output!
  [{:keys [output-binary? output-directory]}
   {:keys [burn-time-matrix flame-length-matrix spread-rate-matrix fire-type-matrix]}
   simulation]
  (when output-binary?
    (let [output-name (format "toa_0001_%05d.bin" (inc simulation))
          output-path (if output-directory
                        (.getPath (io/file output-directory output-name))
                        output-name)]
      (binary/write-matrices-as-binary output-path
                                       [:float :float :float :int]
                                       [(m/emap #(if (pos? %) (* 60 %) %) burn-time-matrix)
                                        flame-length-matrix
                                        spread-rate-matrix
                                        fire-type-matrix]))))

(defn get-envelope
  [config landfire-layers]
  (let [{:keys [upperleftx upperlefty width height scalex scaley]} (landfire-layers :elevation)]
    (make-envelope (:srid config)
                   upperleftx
                   (+ upperlefty (* height scaley))
                   (* width scalex)
                   (* -1.0 height scaley))))

(defn cell-size-multiplier
  [cell-size {:keys [scalex]}]
  (int (quot (m->ft scalex) cell-size)))  ; FIXME: Should we be using /?

;; FIXME: This would be simpler is we flattened fuel-moisture-layers into a single-level map
(defn create-multiplier-lookup
  [{:keys [cell-size weather-layers fuel-moisture-layers]}]
  (let [layers (merge weather-layers fuel-moisture-layers)]
    (reduce (fn [acc ks] (let [layer (get-in layers ks)]
                           (if (map? layer)
                             (assoc-in acc ks (cell-size-multiplier cell-size layer))
                             acc)))
            {}
            [[:temperature]
             [:relative-humidity]
             [:wind-speed-20ft]
             [:wind-from-direction]
             [:dead :1hr]
             [:dead :10hr]
             [:dead :100hr]
             [:live :herbaceous]
             [:live :woody]])))

(defn get-weather [config rand-generator weather-type weather-layers]
  (if (contains? weather-layers weather-type)
    (weather-type weather-layers)
    (draw-samples rand-generator (:simulations config) (config weather-type))))

;; FIXME: Rename :landfire-rasters to :landfire-matrices everywhere (do we have to pass this parameter around?)
(defn add-input-layers
  [config]
  (let [landfire-layers (fetch/landfire-layers config)]
    (assoc config
           :envelope             (get-envelope config landfire-layers)
           :landfire-rasters    (into {}
                                      (map (fn [[layer-name layer-info]] [layer-name (first (:matrix layer-info))]))
                                      landfire-layers)
           :ignition-layer       (fetch/ignition-layer config)
           :ignition-mask-layer  (fetch/ignition-mask-layer config)
           :weather-layers       (fetch/weather-layers config)
           :fuel-moisture-layers (fetch/fuel-moisture-layers config))))

(defn add-misc-params
  [{:keys [random-seed landfire-rasters] :as inputs}]
  (assoc inputs
         :num-rows          (m/row-count (:fuel-model landfire-rasters))
         :num-cols          (m/column-count (:fuel-model landfire-rasters))
         :multiplier-lookup (create-multiplier-lookup inputs)
         :rand-gen          (if random-seed
                              (Random. random-seed)
                              (Random.))))

(defn add-ignition-csv
  [{:keys [ignition-csv] :as inputs}]
  (if ignition-csv
    (let [ignitions (with-open [reader (io/reader ignition-csv)]
                      (doall (rest (csv/read-csv reader))))]
      (assoc inputs
             :ignition-rows        (mapv #(Integer/parseInt (get % 0)) ignitions)
             :ignition-cols        (mapv #(Integer/parseInt (get % 1)) ignitions)
             :ignition-start-times (mapv #(Double/parseDouble (get % 2)) ignitions)
             :max-runtimes         (mapv #(Double/parseDouble (get % 3)) ignitions)
             :simulations          (count ignitions)))
    inputs))

;; FIXME: Try using draw-sample within run-simulation instead of draw-samples here.
(defn add-sampled-params
  [{:keys [rand-gen simulations max-runtime ignition-row ignition-col
           foliar-moisture ellipse-adjustment-factor perturbations ignition-rows
           ignition-cols max-runtimes]
    :as inputs}]
  (assoc inputs
         :max-runtimes               (or max-runtimes (draw-samples rand-gen simulations max-runtime))
         :ignition-rows              (or ignition-rows (draw-samples rand-gen simulations ignition-row))
         :ignition-cols              (or ignition-cols (draw-samples rand-gen simulations ignition-col))
         :foliar-moistures           (draw-samples rand-gen simulations foliar-moisture)
         :ellipse-adjustment-factors (draw-samples rand-gen simulations ellipse-adjustment-factor)
         :perturbations              (perturbation/draw-samples rand-gen simulations perturbations))) ;FIXME: shadowed

;; FIXME: Try using draw-sample within run-simulation instead of get-weather here.
(defn add-weather-params
  [{:keys [rand-gen weather-layers] :as inputs}]
  (assoc inputs
         :temperatures         (get-weather inputs rand-gen :temperature weather-layers)
         :relative-humidities  (get-weather inputs rand-gen :relative-humidity weather-layers)
         :wind-speeds-20ft     (get-weather inputs rand-gen :wind-speed-20ft weather-layers)
         :wind-from-directions (get-weather inputs rand-gen :wind-from-direction weather-layers)))

(defn add-ignitable-sites
  [{:keys [ignition-mask-layer num-rows num-cols] :as inputs}]
  (let [ignition-mask-indices (some->> ignition-mask-layer
                                       :matrix
                                       first
                                       m/non-zero-indices
                                       (map-indexed (fn [i v] (when (pos? (count v)) [i v])))
                                       (filterv identity))
        ignitable-sites (if ignition-mask-indices
                          (for [[row cols] ignition-mask-indices
                                col        cols
                                :when      (random-ignition/valid-ignition-site? inputs row col)]
                            [row col])
                          (for [row   (range num-rows)
                                col   (range num-cols)
                                :when (random-ignition/valid-ignition-site? inputs row col)]
                            [row col]))]
    (assoc inputs :ignitable-sites ignitable-sites)))
```

```
(defn initialize-aggregate-matrices
  [{:keys [num-rows num-cols output-flame-length-sum?
           output-flame-length-max? output-spot-count?] :as inputs}]
  {:burn-count-matrix       (initialize-burn-count-matrix inputs)
   :flame-length-sum-matrix (when output-flame-length-sum? (m/zero-array [num-rows num-cols]))
   :flame-length-max-matrix (when output-flame-length-max? (m/zero-array [num-rows num-cols]))
   :spot-count-matrix       (when output-spot-count? (m/zero-array [num-rows num-cols]))})

(defn add-aggregate-matrices
  [inputs]
  (merge inputs (initialize-aggregate-matrices inputs)))

(defn load-inputs
  [config]
  (-> config
      (add-input-layers)
      (add-misc-params)
      (add-ignition-csv)
      (add-sampled-params)
      (add-weather-params)
      (add-ignitable-sites)
      (add-aggregate-matrices)))

;; FIXME: Replace input-variations expression with add-sampled-params
;;        and add-weather-params (and remove them from load-inputs).
;;        This will require making these function return single
;;        samples instead of sequences of samples. Also combine the
;;        initial-ignition-site calculation into input-variations or
;;        move it to run-fire-spread.
(defn run-simulation!
  [{:keys [output-csvs? envelope ignition-layer cell-size
           max-runtimes ignition-rows ignition-cols foliar-moistures ellipse-adjustment-factors perturbations
           temperatures relative-humidities wind-speeds-20ft wind-from-directions
           random-seed ignition-start-times] :as inputs}
   i]
  (tufte/profile
   {:id :run-simulation}
   (let [matrix-or-i           (fn [obj i] (:matrix obj (obj i)))
         initial-ignition-site (or ignition-layer
                                   (when (and (ignition-rows i) (ignition-cols i))
                                     [(ignition-rows i) (ignition-cols i)]))
         input-variations      {:rand-gen                   (if random-seed (Random. (+ random-seed i)) (Random.))
                                :max-runtime                (max-runtimes i)
                                :foliar-moisture            (* 0.01 (foliar-moistures i))
                                :ellipse-adjustment-factor (ellipse-adjustment-factors i)
                                :perturbations              (when perturbations (perturbations i))
                                :temperature                (matrix-or-i temperatures i)
                                :relative-humidity          (matrix-or-i relative-humidities i)
                                :wind-speed-20ft            (matrix-or-i wind-speeds-20ft i)
                                :wind-from-direction        (matrix-or-i wind-from-directions i)
                                :ignition-start-time        (get ignition-start-times i 0.0)}
         fire-spread-results   (tufte/p :run-fire-spread
                                         (run-fire-spread
                                          (merge inputs
                                                 input-variations
                                                 {:initial-ignition-site initial-ignition-site})))]
     (when fire-spread-results
       (process-output-layers! inputs fire-spread-results envelope i)
       (process-aggregate-output-layers! inputs fire-spread-results)
       (process-binary-output! inputs fire-spread-results i))
     (when output-csvs?
       (merge
        input-variations
        {:simulation      (inc i)
         :ignition-row    (ignition-rows i)
         :ignition-col    (ignition-cols i)
         :foliar-moisture (foliar-moistures i)
         :global-clock    (:global-clock fire-spread-results)
         :exit-condition   (:exit-condition fire-spread-results :no-fire-spread)
         :crown-fire-count (:crown-fire-count fire-spread-results)}
        (if fire-spread-results
          (tufte/p
           :summarize-fire-spread-results
           (summarize-fire-spread-results fire-spread-results cell-size))
          {:fire-size                 0.0
           :flame-length-mean         0.0
           :flame-length-stddev       0.0
           :fire-line-intensity-mean  0.0
           :fire-line-intensity-stddev 0.0}))))))

(defn run-simulations!
  [{:keys [simulations parallel-strategy] :as inputs}]
  (log-str "Running simulations")
  (let [stats-accumulator      (do
                                 (tufte/remove-handler! :accumulating)
                                 (tufte/add-accumulating-handler! {:handler-id :accumulating}))
        parallel-bin-size      (max 1 (quot simulations (.availableProcessors (Runtime/getRuntime))))
        reducer-fn             (if (= parallel-strategy :between-fires)
                                 #(into [] (r/fold parallel-bin-size r/cat r/append! %))
                                 #(into [] %))
        summary-stats          (with-redefs [rothermel-fast-wrapper (memoize rothermel-fast-wrapper)]
                                 (->> (vec (range simulations))
                                      (r/map (partial run-simulation! inputs))
                                      (r/remove nil?)
                                      (reducer-fn)))]
    (Thread/sleep 1000)
    (log (tufte/format-grouped-pstats @stats-accumulator
                                      {:format-pstats-opts {:columns [:n-calls :min :max :mean :mad :clock :total]}})
         :truncate? false)
    {:burn-count-matrix       (:burn-count-matrix inputs)
     :flame-length-sum-matrix (:flame-length-sum-matrix inputs)
     :flame-length-max-matrix (:flame-length-max-matrix inputs)
     :spot-count-matrix       (:spot-count-matrix inputs)
     :summary-stats           summary-stats}))

;;-----------------------------------------------------------------------------
;; Outputs ;TODO move section to it's own ns
;;-----------------------------------------------------------------------------

(defn write-landfire-layers!
  [{:keys [output-landfire-inputs? outfile-suffix landfire-rasters envelope]}]
  (when output-landfire-inputs?
    (doseq [[layer matrix] landfire-rasters]
      (-> (matrix-to-raster (name layer) matrix envelope)
          (write-raster (str (name layer) outfile-suffix ".tif"))))))

(defn write-burn-probability-layer!
  [{:keys [output-burn-probability simulations envelope output-pngs?] :as inputs} {:keys [burn-count-matrix]}]
  (when-let [timestep output-burn-probability]
    (let [output-name "burn_probability"]
      (if (int? timestep)
        (doseq [[band matrix] (map-indexed vector burn-count-matrix)]
          (let [output-time        (* band timestep)
                probability-matrix (m/emap #(/ % simulations) matrix)]
            (output-geotiff inputs probability-matrix output-name envelope nil output-time)
            (output-png inputs probability-matrix output-name envelope nil output-time)))
        (let [probability-matrix (m/emap #(/ % simulations) burn-count-matrix)]
          (output-geotiff inputs probability-matrix output-name envelope)
          (when output-pngs?
           (output-png inputs probability-matrix output-name envelope)))))))

(defn write-flame-length-sum-layer!
  [{:keys [envelope output-flame-length-sum?] :as inputs}
   {:keys [flame-length-sum-matrix]}]
  (when output-flame-length-sum?
    (output-geotiff inputs flame-length-sum-matrix "flame_length_sum" envelope)))
```

```
(defn write-flame-length-max-layer!
  [{:keys [envelope output-flame-length-max?] :as inputs}
   {:keys [flame-length-max-matrix]}]
  (when output-flame-length-max?
    (output-geotiff inputs flame-length-max-matrix "flame_length_max" envelope)))

(defn write-burn-count-layer!
  [{:keys [envelope output-burn-count?] :as inputs}
   {:keys [burn-count-matrix]}]
  (when output-burn-count?
    (output-geotiff inputs burn-count-matrix "burn_count" envelope)))

(defn write-spot-count-layer!
  [{:keys [envelope output-spot-count?] :as inputs}
   {:keys [spot-count-matrix]}]
  (when output-spot-count?
    (output-geotiff inputs spot-count-matrix "spot_count" envelope)))

(defn write-aggregate-layers!
  [inputs outputs]
  (write-burn-probability-layer! inputs outputs)
  (write-flame-length-sum-layer! inputs outputs)
  (write-flame-length-max-layer! inputs outputs)
  (write-burn-count-layer! inputs outputs)
  (write-spot-count-layer! inputs outputs))

(defn write-csv-outputs!
  [{:keys [output-csvs? output-directory outfile-suffix]} {:keys [summary-stats]}]
  (when output-csvs?
    (let [output-filename (str "summary_stats" outfile-suffix ".csv")]
      (with-open [out-file (io/writer (if output-directory
                                        (str/join "/" [output-directory output-filename])
                                        output-filename))]
        (->> summary-stats
             (sort-by #(vector (:ignition-row %) (:ignition-col %)))
             (mapv (fn [{:keys [ignition-row ignition-col max-runtime temperature relative-humidity
                                wind-speed-20ft wind-from-direction foliar-moisture ellipse-adjustment-factor
                                fire-size flame-length-mean flame-length-stddev fire-line-intensity-mean
                                fire-line-intensity-stddev simulation crown-fire-size spot-count]}]
                     [simulation
                      ignition-row
                      ignition-col
                      max-runtime
                      temperature
                      relative-humidity
                      wind-speed-20ft
                      wind-from-direction
                      foliar-moisture
                      ellipse-adjustment-factor
                      fire-size
                      flame-length-mean
                      flame-length-stddev
                      fire-line-intensity-mean
                      fire-line-intensity-stddev
                      crown-fire-size
                      spot-count]))
             (cons ["simulation" "ignition-row" "ignition-col" "max-runtime" "temperature" "relative-humidity"
                    "wind-speed-20ft" "wind-from-direction" "foliar-moisture" "ellipse-adjustment-factor"
                    "fire-size" "flame-length-mean" "flame-length-stddev" "fire-line-intensity-mean"
                    "fire-line-intensity-stddev" "crown-fire-size" "spot-count"])
             (csv/write-csv out-file))))))

(defn process-config-file!
  [config-file]
  (let [config (edn/read-string (slurp config-file))]
    (if-not (spec/valid? ::config-spec/config config)
      (spec/explain ::config-spec/config config)
      (let [inputs (load-inputs config)]
        (if (seq (:ignitable-sites inputs))
          (let [outputs (run-simulations! inputs)]
            (write-landfire-layers! inputs)
            (write-aggregate-layers! inputs outputs)
            (write-csv-outputs! inputs outputs))
          (log-str "Could not run simulation. No valid ignition sites. Config:" config-file))))
    (shutdown-agents)))
```

```
(ns gridfire.cli
  (:gen-class)
  (:require [clojure.core.async    :refer [<!!]]
            [clojure.edn           :as edn]
            [clojure.java.io       :as io]
            [clojure.tools.cli     :refer [parse-opts]]
            [gridfire.config       :as config]
            [gridfire.core         :as gridfire]
            [gridfire.server       :as server]
            [gridfire.utils.server :refer [hostname? throw-message]]))

(set! *unchecked-math* :warn-on-boxed)

;;=============================================================
;; Argument Processing
;;=============================================================

(defn all-required-keys? [arguments options]
  (or (seq arguments)
      (every? options [:server-config :host :port])
      (:elmfire-data options)))

(defn process-options [arguments {:keys [server-config] :as options}]
  (cond (not (all-required-keys? arguments options))
        (throw-message (str "For gridfire cli mode, include "
                            "one or more gridfire.edn files.\n"
                            "For gridfire server mode, include these args: "
                            "--server-config --host --port\n"
                            "For converting elmfire.data to gridfire.edn, include this arg: "
                            "--elmfire-data"))

        server-config
        (let [config-file-params  (edn/read-string (slurp server-config))
              command-line-params (dissoc options :server-config)]
          (merge config-file-params command-line-params))

        :else
        options))

;;=============================================================
;; User Interface
;;=============================================================

(def cli-options
  [["-c" "--server-config CONFIG" "Server config file"
    :validate [#(.exists  (io/file %)) "The provided --server-config does not exist."
               #(.canRead (io/file %)) "The provided --server-config is not readable."]]

   ["-h" "--host HOST" "Host domain name"
    :validate [hostname? "The provided --host is invalid."]]

   ["-p" "--port PORT" "Port number"
    :parse-fn #(if (int? %) % (Integer/parseInt %))
    :validate [#(< 0 % 0x10000) "Must be a number between 0 and 65536"]]

   ["-e" "--elmfire-data FILE" "Path to an elmfire.data file"
    :validate [#(.exists  (io/file %)) "The provided --elmfire-data does not exist."
               #(.canRead (io/file %)) "The provided --elmfire-data is not readable."]]
   ["-v" "--verbose" "Flag for controlling elmfire.data conversion output params"]

   ["-o" "--override-config OVERRIDE" "Path to override.edn file"
    :validate [#(.exists  (io/file %)) "The provided --override-config does not exist."
               #(.canRead (io/file %)) "The provided --override-config is not readable."]]])

(def program-banner
  (str "gridfire: Launch fire spread simulations via config files or in server mode.\n"
       "Copyright © 2014-2021 Spatial Informatics Group, LLC.\n"))

(defn -main
  [& args]
  (println program-banner)
  (let [{:keys [options arguments summary errors]} (parse-opts args cli-options)
        ;; {:options   The options map, keyed by :id, mapped to the parsed value
        ;;  :arguments A vector of unprocessed arguments
        ;;  :summary   A string containing a minimal options summary
        ;;  :errors    A vector of error message strings thrown during parsing; nil when no errors exist
        config-params (try
                        (process-options arguments options)
                        (catch Exception e
                          (ex-message e)))]
    (cond
      (seq errors)
      (do
        (run! println errors)
        (println (str "\nUsage:\n" summary)))

      (string? config-params)
      (do
        (println config-params)
        (println (str "\nUsage:\n" summary)))

      (:elmfire-data options)
      (config/convert-config! (:elmfire-data options) (:override-config options))

      (:server-config options)
      (<!! (server/start-server! config-params))

      :else
      (doseq [config-file arguments]
        (gridfire/process-config-file! config-file)))
    (System/exit 0)))
```

```
(ns gridfire.utils.random
  (:import java.util.Random))

(defn my-rand
  ([^Random rand-generator] (.nextDouble rand-generator))
  (^double [^Random rand-generator n] (* n (my-rand rand-generator))))

(defn my-rand-int
  ^long
  [rand-generator n]
  (int (my-rand rand-generator n)))

(defn my-rand-nth
  [rand-generator coll]
  (nth coll (my-rand-int rand-generator (count coll))))

;; FIXME: This function is redundant with my-rand-range and can be removed.
(defn random-float
  [min-val max-val rand-generator]
  (let [range (- max-val min-val)]
    (+ min-val (my-rand rand-generator range))))
```

```
(defn my-rand-range
  [rand-generator [min-val max-val]]
  (let [range (- max-val min-val)]
    (+ min-val (if (int? range)
                 (my-rand-int rand-generator range)
                 (my-rand rand-generator range)))))

(defn sample-from-list
  [rand-generator n xs]
  (repeatedly n #(my-rand-nth rand-generator xs)))

(defn sample-from-range
  [rand-generator n [min-val max-val]]
  (repeatedly n #(my-rand-range rand-generator [min-val max-val])))

(defn draw-sample
  [rand-generator x]
  (cond (list? x)   (my-rand-nth rand-generator x)
        (vector? x) (my-rand-range rand-generator x)
        :else       x))

(defn draw-samples
  [rand-generator n x]
  (into []
        (cond (list? x)   (sample-from-list rand-generator n x)
              (vector? x) (sample-from-range rand-generator n x)
              :else       (repeat n x))))
```

6.4 Server Interface

The GridFire system is also available for use in server mode. The GridFire server will listen for requests to launch fire sparead simulations. Upon completion of the simulation a set of post processing scripts will run to process binary outputs into a directory structure containing geoTIFF files and then packed into a tarball. This tarball is sent over scp into another server for processing.

6.4.1 Post processing dependencies

The Post processing scripts require the following packages to work:

- ssh
- pigz
- mpirun
- gdal
- elmfire_post_¡elmfire_version¿ (for elmfire version see ‘resources/elmfire_post.sh‘)

6.4.2 Server Configuration

The GridFire system is also available for use in server mode. The server is confgured by and edn file containing the following contents:

```
{:software-dir    "/gridfire/software"
 :incoming-dir    "/gridfire/incoming"
 :active-fire-dir "/gridfire/incoming/active_fires"
 :data-dir        "/gridfire/data"
 :log-dir         "/gridfire/log"}
```

software-dir The directory the gridfire repo is cloned into.

incoming-dir The directory where the server will look for match-drop input decks (which should be uploaded from the data provisioning server on wx.pyregence.org). The server keeps the latest 20 input decks. (see cleanup.sh in this directory)

active-fire-dir The directory where the server will look for active-fire input decks (which are uploaded from the data provisioning server on wx.pyregence.org). The server's file watcher thread will automatically add a request onto the server's 'standby-queue' whenever a new input deck is uploaded. The server keeps the latest 200 input decks. (see cleanup.sh in this directory)

data-dir The directory into which input deck tarballs from incoming-dir are unpacked. The server keeps the latest 20 unpacked tarballs. (see cleanup.sh in this directory)

log-dir The directory into which log files are written. The server keeps the last 10 days of logs.

The GridFire server is launched by user 'gridfire' with this command:

```
clojure -M:run-server -c server.edn
```

7 Configuration File

The configuration file for GridFire's command line interface is a text file in Extensible Data Notation (EDN) format.[5] A sample configuration file is provided below and in "resources/sample_config.edn". The format should be self-evident at a glance, but it is worth noting that EDN is case-sensitive but whitespace-insensitive. Comments are anything following two semi-colons (;;). Strings are contained in double-quotes (""). Keywords are prefixed with a colon (:). Vectors are delimited with square brackets ([]). Associative lookup tables (a.k.a. maps) are delimited with curly braces ({}) and are used to express key-value relationships.

The configuration file can be broken up into 5 sections as described below:

7.1 Section 1: Landscape data to be shared by all simulations

GridFire allows us to choose how we want to ingest landscape data through the configuration file. We can choose to get LANDFIRE layers from our PostGIS database, or we can read raster files from disk. This behavior is controlled as follows:

Include the following mapping at the top level of the configuraiton file:

- **landfire-layers**: a map of fetch specifications

For the fetch specifications include the following mappings:

- **type**: the method for fetching the layer
- **source**: the string input for the fetch method

To fetch layers from a Postgresql database you must also include the follwing mapping:

- **db-spec**: a map of database connection information for our Postgresql database

[5] https://github.com/edn-format/edn

Here's an example of fetching LANDFIRE layers from a Postgresql database.

```
{:db-spec             {:classname   "org.postgresql.Driver"
                       :subprotocol "postgresql"
                       :subname     "//localhost:5432/gridfire"
                       :user        "gridfire"}
 :landfire-layers     {:aspect             {:type   :postgis
                                            :source "landfire.asp WHERE rid=100"}
                       :canopy-base-height {:type   :postgis
                                            :source "landfire.cbh WHERE rid=100"}
                       :canopy-cover       {:type   :postgis
                                            :source "landfire.cc WHERE rid=100"}
                       :canopy-height      {:type   :postgis
                                            :source "landfire.ch WHERE rid=100"}
                       :crown-bulk-density {:type   :postgis
                                            :source "landfire.cbd WHERE rid=100"}
                       :elevation          {:type   :postgis
                                            :source "landfire.fbfm40 WHERE rid=100"}
                       :fuel-model         {:type   :postgis
                                            :source "landfire.slp WHERE rid=100"}
                       :slope              {:type   :postgis
                                            :source "landfire.dem WHERE rid=100"}}}
```

Here's an example of fetching LANDFIRE layers from files on disk.

```
{:landfire-layers {:aspect             {:type   :geotiff
                                        :source "test/gridfire/resources/asp.tif"}
                   :canopy-base-height {:type   :geotiff
                                        :source "test/gridfire/resources/cbh.tif"}
                   :canopy-cover       {:type   :geotiff
                                        :source "test/gridfire/resources/cc.tif"}
                   :canopy-height      {:type   :geotiff
                                        :source "test/gridfire/resources/ch.tif"}
                   :crown-bulk-density {:type   :geotiff
                                        :source "test/gridfire/resources/cbd.tif"}
                   :elevation          {:type   :geotiff
                                        :source "test/gridfire/resources/dem.tif"}
                   :fuel-model         {:type   :geotiff
                                        :source "test/gridfire/resources/fbfm40.tif"}
                   :slope              {:type   :geotiff
                                        :source "test/gridfire/resources/slp.tif"}}}
```

Gridfire uses imperial units for its calculations. Gridfire optionally allows us to use LANDFIRE LAYERS in different units and scale.

To specify the need for conversion from metric to imperial, include the following mapping in the fetch specifications:

- **units**: keyword :metric

```
{:canopy-height {:type   :geotiff
                 :source "test/gridfire/resources/weather-test/ch.tif"
                 :units  :metric}}
```

To specify a scaling factor, include the following mapping in the fetch specifications:

- **multiplier**: int or float

```
{:canopy-height {:type       :geotiff
                 :source     "test/gridfire/resources/weather-test/ch.tif"
                 :multiplier 0.1}}
```

In the example above the input raster's units are meters * 10.[6] Thus a value of 5 on the canopy height grid layer is actually 0.5 meters. The mutipler factor needed to convert to meters is 0.1.

Include the following required mapping on all configurations:

```
{:srid      "CUSTOM:900914"
 :cell-size 98.425} ; (feet)
```

7.2 Section 2: Ignition data from which to build simulation inputs

GridFire allows us to choose how we want to initialize the ignition area. We can choose one of 2 options: to initialize a single point or an existing burn perimeter (raster).

To initialize a single point, include the following mappings:

- **ignition-row**: (single, list, or range of values)
- **ignition-col**: (single, list, or range of values)

For this method of ignition, values may be entered in one of three ways:

1. If a single value is provided, it will be kept the same for all simulations.
2. For a list of values, a value from the list will be randomly selected in each simulation.
3. For a range of values, a value from the range [inclusive exclusive] will be randomly selected in each simulation.

```
{:ignition-row [10 90]
 :ignition-col [20 80]}
```

Another way we can ignite a single point is to omit the keys **ignition-row** and **ignition-col**. With this method, we can optionally constrain the ignition location by an ignition-mask raster and/or an edge-buffer. For specifiying these constraints include these optional mappings:

- **ignition-mask**: a map of fetch specifications (see section 1)
- **edge-buffer**: the thickness (feet) along the edge of the computational domain where ignitions cannot occur

Note: Nonzero values in the ignition mask are considered ignitable

Here's an example of specifiying ignition points using an ignition mask from a geotiff file.

```
{:ignition-mask {:raster {:type :geotiff
                          :source "test/gridfire/resources/weather-test/ignition-mask.tif"}
                 :edge-buffer 98.4}}
```

Note: ignition-row and ignition-col must be omitted for this feature.

Here's the namespace that implments this functionality.

[6] https://landfire.gov/faqprint.php

```clojure
(ns gridfire.random-ignition
  (:require [clojure.core.matrix :as m]
            [gridfire.common     :refer [burnable-fuel-model?]]))

(m/set-current-implementation :vectorz)

(defn- in-edge-buffer?
  "Returns true if give [row col] is within the buffer region defined
  by buffer-size. The buffer size is the thickness (pixel) of the edge
  buffer region."
  [num-rows num-cols buffer-size row col]
  (or  (<= row buffer-size)
       (> row (- num-rows buffer-size))
       (<= col buffer-size)
       (> col (- num-cols buffer-size))))

(defn valid-ignition-site? [{:keys [num-rows num-cols random-ignition cell-size landfire-rasters]} row col]
  (let [{:keys [edge-buffer]} random-ignition
        {:keys [fuel-model]}  landfire-rasters]
    (and (if edge-buffer
           (let [buffer-size (int (Math/ceil (/ edge-buffer cell-size)))]
             (not (in-edge-buffer? num-rows num-cols buffer-size row col)))
           true)
         (burnable-fuel-model? (m/mget fuel-model row col)))))
```

To initialize an existing burn perimeter from a raster, we have two options. We can read rasters from a Postgresql database or a raster file on disk. This behavior is controlled as follows:
Include the following mapping at the top level of the configuraiton file:

- **ignition-layer**: a map of fetch specifications

For the fetch specifications include the following mappings:

- **type**: the method for fetching the layer
- **source**: the string input for the fetch method

Here's an example of fetching an intial burn perimeter from a Postgresql database.
,*Note*: be sure to include the map of database connection (**:db-spec**) as described in section 1. ,#+begin_src clojure {:ignition-layer {:type :postgis :source "ignition.ign WHERE rid=1"}} #+end_src
Here's an example of fetching an intial burn perimeter from a file on disk

```clojure
{:fetch-ignition-method :geotiff
 :ignition-layer        "test/gridfire/resources/ign.tif"}
```

GridFire makes use of clojure's multimethods to dispatch control to different handlers for fetching ignition layers. The dispatch depends on what is in the config file. Here's the namespace that implements this functionality.

```clojure
(ns gridfire.fetch
  (:require [clojure.core.matrix      :as m]
            [gridfire.conversion      :as convert]
            [gridfire.magellan-bridge :refer [geotiff-raster-to-matrix]]
            [gridfire.postgis-bridge  :refer [postgis-raster-to-matrix]]))

(m/set-current-implementation :vectorz)

;;TODO refactor multi-methods landfire-layer weather, ignition-layer, ignition-mask-layer
;; to the same function
```

```
;;-----------------------------------------------------------------------------
;; LANDFIRE
;;-----------------------------------------------------------------------------

(def layer-names
  [:aspect
   :canopy-base-height
   :canopy-cover
   :canopy-height
   :crown-bulk-density
   :elevation
   :fuel-model
   :slope])

(defmulti landfire-layer
  (fn [_ {:keys [type]}] type))

(defmethod landfire-layer :postgis
  [db-spec {:keys [source]}]
  (postgis-raster-to-matrix db-spec source))

(defmethod landfire-layer :geotiff
  [_ {:keys [source]}]
  (geotiff-raster-to-matrix source))

(defn landfire-layers
  "Returns a map of LANDFIRE layers (represented as maps) with the following units:
   {:elevation          feet
    :slope              vertical feet/horizontal feet
    :aspect             degrees clockwise from north
    :fuel-model         fuel model numbers 1-256
    :canopy-height      feet
    :canopy-base-height feet
    :crown-bulk-density lb/ft^3
    :canopy-cover       % (0-100)}"
  [{:keys [db-spec] :as config}]
  (into {}
        (map (fn [layer-name]
               (let [source (get-in config [:landfire-layers layer-name])]
                 [layer-name
                  (if (map? source)
                    (-> (landfire-layer db-spec source)
                        (convert/to-imperial! source layer-name))
                    (-> (postgis-raster-to-matrix db-spec source)
                        (convert/to-imperial! {:units :metric} layer-name)))]))
        layer-names))

;;-----------------------------------------------------------------------------
;; Initial Ignition
;;-----------------------------------------------------------------------------

(defn convert-burn-values [matrix {:keys [burned unburned]}]
  (m/emap #(condp = %
             (double burned)   1.0
             (double unburned) 0.0
             -1.0)
          matrix))

(defmulti ignition-layer
  (fn [{:keys [ignition-layer]}] (:type ignition-layer)))

(defmethod ignition-layer :postgis
  [{:keys [db-spec ignition-layer]}]
  (let [layer (postgis-raster-to-matrix db-spec (:source ignition-layer))]
    (if-let [burn-values (:burn-values ignition-layer)]
      (update layer :matrix convert-burn-values burn-values)
      layer)))
```

```
(defmethod ignition-layer :geotiff
  [{:keys [ignition-layer]}]
  (let [layer (geotiff-raster-to-matrix (:source ignition-layer))]
    (if-let [burn-values (:burn-values ignition-layer)]
      (update layer :matrix convert-burn-values burn-values)
      layer)))

(defmethod ignition-layer :default
  [_]
  nil)

;;-----------------------------------------------------------------------------
;; Weather
;;-----------------------------------------------------------------------------

(def weather-names
  [:temperature :relative-humidity :wind-speed-20ft :wind-from-direction])

(defmulti weather
  (fn [_ {:keys [type]}] type))

(defmethod weather :postgis
  [{:keys [db-spec]} {:keys [source]}]
  (postgis-raster-to-matrix db-spec source))

(defmethod weather :geotiff
  [_ {:keys [source]}]
  (geotiff-raster-to-matrix source))

(defn weather-layers
  "Returns a map of weather layers (represented as maps) with the following units:
   {:temperature         farenheight
    :relative-humidity   %
    :wind-speed-20ft     mph
    :wind-from-direction degrees clockwise from north}"
  [config]
  (into {}
        (map (fn [weather-name]
               (let [weather-spec (get config weather-name)]
                 (when (map? weather-spec)
                   [weather-name (-> (weather config weather-spec)
                                     (convert/to-imperial! weather-spec weather-name))])))
        weather-names))

;;-----------------------------------------------------------------------------
;; Ignition Mask
;;-----------------------------------------------------------------------------

(defmulti ignition-mask-layer
  (fn [{:keys [random-ignition]}]
    (when (map? random-ignition)
      (get-in random-ignition [:ignition-mask :type]))))

(defmethod ignition-mask-layer :geotiff
  [{:keys [random-ignition]}]
  (geotiff-raster-to-matrix (get-in random-ignition [:ignition-mask :source])))

(defmethod ignition-mask-layer :postgis
  [{:keys [db-spec random-ignition]}]
  (postgis-raster-to-matrix db-spec (get-in random-ignition [:ignition-mask :source])))

(defmethod ignition-mask-layer :default [_] nil)

;;-----------------------------------------------------------------------------
;; Moisture Layers
;;-----------------------------------------------------------------------------
```

```
(defmulti fuel-moisture-layer
  (fn [_ {:keys [type]}] type))

(defmethod fuel-moisture-layer :geotiff
  [_ {:keys [source]}]
  (geotiff-raster-to-matrix source))

(defmethod fuel-moisture-layer :postgis
  [db-spec {:keys [source]}]
  (postgis-raster-to-matrix db-spec source))

(defn fuel-moisture-layers
  "Returns a map of moisture layers (represented as maps) with the following units:
  {:dead {:1hr        %
          :10hr       %
          :100hr      %
   :live {:herbaceous %
          :woody      %"
  [{:keys [db-spec fuel-moisture-layers]}]
  (when fuel-moisture-layers
    (letfn [(f [fuel-class]
              (fn [spec]
                (cond (and (map? spec) (= fuel-class :live))
                      (-> (fuel-moisture-layer db-spec spec)
                          (update :matrix (comp first (fn [m] (m/emap #(* % 0.01) m)))))

                      (map? spec)
                      (-> (fuel-moisture-layer db-spec spec)
                          (update :matrix (fn [m] (m/emap #(* % 0.01) m))))

                      :else
                      (* spec 0.01))))]
      (-> fuel-moisture-layers
          (update-in [:dead :1hr] (f :dead))
          (update-in [:dead :10hr] (f :dead))
          (update-in [:dead :100hr] (f :dead))
          (update-in [:live :herbaceous] (f :live))
          (update-in [:live :woody] (f :live))))))
```

7.3 Section 3: Weather data from which to build simulation inputs

For all the options in this section, you may enter values in one of three ways (as described in section 2): single, list, or range of values.

```
{:temperature                (50 65 80)     ; (degrees Fahrenheit)
 :relative-humidity          (1 10 20)      ; (%)
 :wind-speed-20ft            (10 15 20)     ; (miles/hour)
 :wind-from-direction        (0 90 180 270) ; (degrees clockwise from north)
 :foliar-moisture            90}            ; (%)
```

Temperature, relative humidity, wind speed, and wind direction accepts an additional type of input. GridFire allows us to use weather data from rasters. To use weather data from raster we have two options. This behavior is controlled as follows:

Include the following mapping at the top level of the configuraiton file:

- [**weather-type**]: a map of fetch specifications

For the fetch specifications include the following mappings:

- **type**: the method for fetching the layer

- **source**: the string input for the fetch method

Here's an example of fetching weather rasters from a Postgresql database. **Note**: be sure to include the map of database connection (**:db-spec**) as described in section 1.

```
{:temperature         {:type   :postgis
                       :source "weather.tmpf WHERE rid=100"}
 :relative-humidity   {:type   :postgis
                       :source "weather.rh WHERE rid=100"}
 :wind-speed-20ft     {:type   :postgis
                       :source "weather.ws WHERE rid=100"}
 :wind-from-direction {:type   :postgis
                       :source "weather.wd WHERE rid=100"}}
```

Here's an example of fetching weather rasters from files on disk.

```
{:temperature         {:type   :geotiff
                       :source "test/gridfire/resources/weather-test/tmpf_to_sample.tif"}
 :relative-humidity   {:type   :geotiff
                       :source "test/gridfire/resources/weather-test/rh_to_sample.tif"}
 :wind-speed-20ft     {:type   :geotiff
                       :source "test/gridfire/resources/weather-test/ws_to_sample.tif"}
 :wind-from-direction {:type   :geotiff
                       :source "test/gridfire/resources/weather-test/d_to_sample.tif"}}
```

NOTE: Gridfire expects weather raster's resolution and the landfire's resolution as designated by the ':cell-size' must be exact multiples of one another. This means you may choose to use raster's of different cell sizes to improve preformance.

Gridfire uses imperial units for its calculations. Gridfire optionally allows us to use weather in different units and scale.

To specify the need for conversion from metric to imperial, include the following mapping in the fetch specifications:

- **units**: keyword :metric

To specify the need for conversion from absolute to imperial, include

- **units**: keyword :absolute

```
{:temperture {:type   :geotiff
              :source "test/gridfire/resources/weather-test/tmpf_to_sample.tif"
              :units  :metric}}
```

7.4 Section 4: Number of simulations and (optional) random seed perimeter

```
{:max-runtime                60             ; (minutes)
 :simulations                10
 :ellipse-adjustment-factor 1.0             ; (< 1.0 = more circular, > 1.0 = more elliptical)
 :random-seed 1234567890}                   ; long value (optional)
```

7.5 Section 5: Outputs

Currently supported Geotiff layers for output

- fire-spread

- flame-length
- fire-line-intensity
- burn-history

To control the layers to output include the following mappings:

- **output-layers**: map of layers-name to timestep (in minutes) or the keyword ':final'
- **output-geotiff**: boolean

```
{:output-layers  {:fire-spread  10
                  :burn-history :final}
 :output-geotiff true}
```

The configuration above specify that we'd like to output one firespread geotiff every 10 minutes in the simulation. For the burn history we'd like to output the geotiff file at the final timestep of the simulation.

Note: if entry for ':output-layers' is omitted but ':output-geotiff' is set to true then Gridfire will output all layers above at the final timestep.

Gridfire also supports a number of layers that aggregate data across simulations.

To control the output of the burn probability layer, which is calcualted as the number of times a cell burned divided by the number of simulations, include the following mapping:

- **output-*burn-probability**: timestep (in minutes) or keyword ':final'

```
{:output-burn-probability 10}
```

To specify the output of the flame length sum layer, which is the sum of flame lengths across simulations, include the following mapping:

- **output-*flame-length-sum**: bolean

```
{:output-flame-length-sum true}
```

To specify the output of the flame length max layer, which is the max of flame lengths across simulations, include the following mapping:

- **output-*flame-length-max**: bolean

```
{:output-flame-length-max true}
```

To specify the output of the burn count layer, which is the number of times a cell has burned across simulations, include the following mapping:

- **output-*burn-count**: bolean

```
{:output-burn-count true}
```

To specify the output of the spot count layer, which is the number of times a spot ignition occured in a cell, include the following mapping:

- **output-*spot-count**: bolean

```
{:output-spot-count true}
```

Other output mappings:

```
{:outfile-suffix          "_tile_100"
 :output-landfire-inputs? true
 :output-pngs?            true
 :output-csvs?            true}
```

7.6 Section 6: Perturbations

Gridfire supports puturbations of input rasters during simulations in order to account for inherent uncertainty in the data. A uniform random sampling of values within a given range is used to address these uncertanties.

To specify this in the config file include the following mappings:

- **perturbations:** a map of layer names to a map of perturbation configurations

```
{:perturbations {:canopy-height {:spatial-type :global
                                 :range        [-1.0 1.0]}}}
```

The above config specify that a randomly selected value between -1.0 and 1.0 should be added to the canopy height value. This perturbation will be applied globally to all cells. We could also, instead, specify that each cell should be perturbed individually by setting **:spatial-type** to **:pixel**.

Gridfire expects perturbations to be in imperial units. If these perturbations are meant to be in metric, you must include an entry for the units:

```
{:perturbations {:canopy-height {:spatial-type :global
                                 :range        [-1.0 1.0]
                                 :units        :metric}}}
```

```
(ns gridfire.perturbation
  (:require [gridfire.utils.random :refer [my-rand-range]]
            [gridfire.conversion :refer [conversion-table]]))

(defn add-rand-generator
  [rand-generator config]
  (into config
        (map (fn [[layer spec]] [layer (assoc spec :rand-generator rand-generator)]))
        config))

(defn add-simulation-id
  [id config]
  (into config
        (map (fn [[layer spec]] [layer (assoc spec :simulation-id id)]))
        config))

(defn convert-ranges
  [config]
  (into config
        (map (fn [[layer {:keys [units range] :as spec}]]
               (if-let [converter (get-in conversion-table [layer units])]
                 [layer (assoc spec :range (map converter range))]
                 [layer spec])))
        config))
```

```
(defn add-global-values
  [config]
  (into config
        (map (fn [[layer {:keys [spatial-type rand-generator range] :as spec}]]
               (if (= spatial-type :global)
                 [layer (assoc spec :global-value (my-rand-range rand-generator range))]
                 [layer spec])))
        config))

(defn- enrich-info
  [perturbations rand-generator id]
  (->> perturbations
      (add-rand-generator rand-generator)
      (add-simulation-id id)
      (convert-ranges)
      (add-global-values)))

(defn draw-samples
  [rand-generator n perturbations]
  (when perturbations
    (mapv #(enrich-info perturbations rand-generator %) (range n))))

(defn value-at
  (^double [perturb-info raster here]
   (value-at perturb-info raster here nil))

  (^double [{:keys [range spatial-type global-value rand-generator]} raster here frequency-band]
   (if (= spatial-type :global)
     global-value
     (my-rand-range rand-generator range))))

(defn- update?
  [^double global-clock ^double next-clock ^long frequency]
  (< (quot global-clock frequency)
     (quot next-clock frequency)))

(defn- global-temporal-perturbations
  [perturbations]
  (->> perturbations
       (filter (fn [[_ v]] (and (:frequency v) (= (:spatial-type v) :global))))
       keys))

(defn update-global-vals
  [{:keys [perturbations] :as constants} current-clock next-clock]
  (let [layers-to-update (global-temporal-perturbations perturbations)]
    (reduce
     (fn [acc layer-name]
       (let [{:keys [frequency
                     range
                     rand-generator]} (get-in acc [:perturbations layer-name])
             new-global               (my-rand-range rand-generator range)]
         (if (update? current-clock next-clock frequency)
           (assoc-in acc [:perturbations layer-name :global-value] new-global)
           acc)))
     constants
     layers-to-update)))
```

7.7 Section 7: Spotting

Gridfire supports spot fires. To turn on spot ignitions include the key **spotting** at the top level of the config file. The value is a map containg these required entries:

- **num-firebrands**: number of firebrands each torched tree will produce

- **decay-constant**: positive number
- **crown-fire-spotting-percent**: probability a crown fire ignition will spot fires

You may also choose to include surface fire spotting. This behavior is controlled by including the following mappings under the **:spotting** configuration:

- **surface-fire-spotting**: a map containing these required entries:
  - **spotting-percent**: a vector of fuel range and percent pairs where the fuel range is a tuple of integers representing the fuel model numbers and the percent is the percentage of surface fire igntion events that will spot fires
  - **critical-fire-line-intensity**: the fireline intensity below which surface fire spotting does not occur

```
{:spotting {:num-firebrands              [10 50]
            :decay-constant              0.005
            :crown-fire-spotting-percent 0.
            :surface-fires-spotting      {:spotting-percent             [[1 100] 1.0]
                                          :critical-fire-line-intensity 2000}}} ;(kW/m)
```

7.8 Section 8: Fuel moisture data from which to build simulation inputs

GridFire allows us to optinally use fuel moisture from rasters instead of calculating it (by default) from temperature and relative humidity. To specifiy this feature include the following mappings at the top level of the config file:

- **fuel-moisture-layers**: a map of fuel moisture specicfiations

```
{:fuel-moisture-layers {:dead {:1hr   {:type   :geotiff
                                       :source "test/gridfire/resources/weather-test/m1_to_sample.tif"}
                               :10hr  {:type   :geotiff
                                       :source "test/gridfire/resources/weather-test/m10_to_sample.tif"}
                               :100hr {:type   :geotiff
                                       :source "test/gridfire/resources/weather-test/m100_to_sample.tif"}}
                        :live {:woody      {:type   :geotiff
                                            :source "test/gridfire/resources/weather-test/mlw_to_sample.tif"}
                               :herbaceous {:type   :geotiff
                                            :source "test/gridfire/resources/weather-test/mlh_to_sample.tif"}}}}
```

Note: Dead fuel moistures are expected to be multiband rasters and live fuel moistures are singleband.

7.9 Section 9: Optimization

Gridifre allows us use multithread processing to improve performance of the simulation run. To specify the type of parallel strategy we'd like to use include the following mapping at the top level of the config file:

- **parallel-strategy**: a keyword of the stratgey

```
{:parallel-strategy :within-fires}
```

Gridfire supports two types of paralleliztion. To prallelize:

- between ensemble of simulations use the keyword **:between-fires**
- within each ensemble member use the keyword **:within-fires**

7.10 Section 10: Spread Algorithm

Gridfire uses the method of adaptive timestep and fractional distances cording to Morais2001. Gridfire provides implementation for two interpretations on how fractional distances are handled. When calculating the fractional distance of a cell, there are 8 possible trajectories from which fire can spread into the cell. When fractional distances are preserved between temesteps, they can be individually tracked so that each trajectory does not have an effect on another, or combined using the largest value among the trajectories. By default Gridfire will track fractional distances individually.

To specify trajectories to have a cumulative effect include the following key value pair at the top level of the config file:

- **:fractional-distance-combination** : the keyword ‘:sum‘

8 Example Configuration files

Here is a complete sample configuration for using landfire layers from our postigs enabled database and initializing burn points from a range of values.

Here is a complete sample configuration for using LANDFIRE layers from our PostGIS-enabled database with ignition points randomly sampled from a range.

```
{;; Section 1: Landscape data to be shared by all simulations
 :db-spec                   {:classname   "org.postgresql.Driver"
                             :subprotocol "postgresql"
                             :subname     "//localhost:5432/gridfire"
                             :user        "gridfire"
                             :password    "gridfire"}
 :landfire-layers            {:aspect             {:type   :postgis
                                                    :source "landfire.asp WHERE rid=100"}
                              :canopy-base-height {:type   :postgis
                                                    :source "landfire.cbh WHERE rid=100"}
                              :canopy-cover       {:type   :postgis
                                                    :source "landfire.cc WHERE rid=100"}
                              :canopy-height      {:type   :postgis
                                                    :source "landfire.ch WHERE rid=100"}
                              :crown-bulk-density {:type   :postgis
                                                    :source "landfire.cbd WHERE rid=100"}
                              :elevation          {:type   :postgis
                                                    :source "landfire.dem WHERE rid=100"}
                              :fuel-model         {:type   :postgis
                                                    :source "landfire.fbfm40 WHERE rid=100"}
                              :slope              {:type   :postgis
                                                    :source "landfire.slp WHERE rid=100"}}
 :srid                      "CUSTOM:900914"
 :cell-size                 98.425         ; (feet)

 ;; Section 2: Ignition data from which to build simulation inputs
 :ignition-row              [10 90]
 :ignition-col              [20 80]

 ;; Section 3: Weather data from which to build simulation inputs
 ;; For all options in this section, you may enter values in one of five ways:
 ;;   1. Single Value: 25
 ;;   2. List of Values: (2 17 9)
 ;;   3. Range of Values: [10 20]
 ;;   4. Raster from file on disk: {:type :geotiff :source "path/to/file/weather.tif"}
 ;;   5. Raster from Postgresql database: {:type :postgis :source "weather.ws WHERE rid=1"}
 ;;
 ;; If a single value is provided, it will be kept the same for all simulations.
 ;; For a list of values, the list will be randomly sampled from in each simulation.
 ;; For a range of values, the range [inclusive exclusive] will be randomly sampled from in each simulation.
 :temperature               (50 65 80)     ; (degrees Fahrenheit)
 :relative-humidity         (1 10 20)      ; (%)
 :wind-speed-20ft           (10 15 20)     ; (miles/hour)
 :wind-from-direction       (0 90 180 270) ; (degrees clockwise from north)
 :foliar-moisture           90             ; (%)

 ;; Section 4: Number of simulations and (optional) random seed parameter
 :max-runtime               60             ; (minutes)
 :ellipse-adjustment-factor 1.0            ; (< 1.0 = more circular, > 1.0 = more elliptical)
 :simulations               10
 :random-seed               1234567890     ; long value (optional)

 ;; Section 5: Types and names of outputs
 :outfile-suffix            "_tile_100"
 :output-landfire-inputs?   true
 :output-geotiffs?          true
 :output-pngs?              true
 :output-csvs?              true}
```

Here is a complete sample configuration for reading both the LANDFIRE layers, initial burn perimeter, and weather layers from GeoTIFF files on disk.

```
{;; Section 1: Landscape data to be shared by all simulations
 :landfire-layers           {:aspect             {:type   :geotiff
                                                  :source "test/gridfire/resources/asp.tif"}
                             :canopy-base-height {:type   :geotiff
                                                  :source "test/gridfire/resources/cbh.tif"}
                             :canopy-cover       {:type   :geotiff
                                                  :source "test/gridfire/resources/cc.tif"}
                             :canopy-height      {:type   :geotiff
                                                  :source "test/gridfire/resources/ch.tif"}
                             :crown-bulk-density {:type   :geotiff
                                                  :source "test/gridfire/resources/cbd.tif"}
                             :elevation          {:type   :geotiff
                                                  :source "test/gridfire/resources/dem.tif"}
                             :fuel-model         {:type   :geotiff
                                                  :source "test/gridfire/resources/fbfm40.tif"}
                             :slope              {:type   :geotiff
                                                  :source "test/gridfire/resources/slp.tif"}}
 :srid                      "CUSTOM:900914"
 :cell-size                 98.425         ; (feet)

 ;; Section 2: Ignition data from which to build simulation inputs
 :ignition-layer            {:type   :geotiff
                             :source "test/gridfire/resources/ign.tif"}

 ;; Section 3: Weather data from which to build simulation inputs
 ;; For all options in this section, you may enter values in one of five ways:
 ;;   1. Single Value: 25
 ;;   2. List of Values: (2 17 9)
 ;;   3. Range of Values: [10 20]
 ;;   4. Raster from file on disk: {:type :geotiff :source "path/to/file/weather.tif"}
 ;;   5. Raster from Postgresql database: {:type :postgis :source "weather.ws WHERE rid=1"}
 ;;
 ;; If a single value is provided, it will be kept the same for all simulations.
 ;; For a list of values, the list will be randomly sampled from in each simulation.
 ;; For a range of values, the range [inclusive exclusive] will be randomly sampled from in each simulation.

 :temperature         {:type   :geotiff
                       :source "test/gridfire/resources/weather-test/tmpf_to_sample.tif"} ; (degrees Fahrenheit)
 :relative-humidity   {:type   :geotiff
                       :source "test/gridfire/resources/weather-test/rh_to_sample.tif"}   ; (%)
 :wind-speed-20ft     {:type   :geotiff
                       :source "test/gridfire/resources/weather-test/ws_to_sample.tif"}   ; (miles/hour)
 :wind-from-direction {:type   :geotiff
                       :source "test/gridfire/resources/weather-test/wd_to_sample.tif"}   ; (degrees cw from north)
 :foliar-moisture     90                                                                   ; (%)

 ;; Section 4: Number of simulations and (optional) random seed parameter
 :max-runtime               60            ; (minutes)
 :ellipse-adjustment-factor 1.0           ; (< 1.0 = more circular, > 1.0 = more elliptical)
 :simulations               10
 :random-seed               1234567890    ; long value (optional)

 ;; Section 5: Types and names of outputs
 :outfile-suffix            "_from_raster_ignition"
 :output-landfire-inputs?   true
 :output-geotiffs?          true
 :output-pngs?              true
 :output-csvs?              true}
```

This concludes our discussion of GridFire's command line interface.

References

Frank A Albini. Estimating wildfire behavior and effects. *General technical report/Intermountain forest and range experiment station. USDA (no. INT-30)*, 1976.

Frank A Albini and Robert G Baughman. Estimating windspeeds for predicting wildland fire behavior. *USDA Forest Service Research Paper INT (USA)*, 1979.

Frank A Albini and Carolyn H Chase. *Fire containment equations for pocket calculators*, volume 268. Intermountain Forest and Range Experiment Station, Forest Service, US Dept. of Agriculture, 1980.

Hal E Anderson. Heat transfer and fire spread. *USDA forest service research paper. Intermountain and range experiment station*, (69), 1969.

Hal E Anderson. Aids to determining fuel models for estimating fire behavior. *The Bark Beetles, Fuels, and Fire Bibliography*, page 143, 1982.

Patricia L Andrews et al. Modeling wind adjustment factor and midflame wind speed for rothermel's surface fire spread model. 2012.

Robert E Burgan. Estimating live fuel moisture for the 1978 national fire danger rating system. *USDA Forest Service Research Paper INT (USA). no. 226.*, 1979.

George M Byram. Combustion of forest fuels. *Forest fire: Control and use*, 1:61–89, 1959.

Miguel G Cruz, Martin E Alexander, and Ronald H Wakimoto. Development and testing of models for predicting crown fire rate of spread in conifer forest stands. *Canadian Journal of Forest Research*, 35 (7):1626–1639, 2005.

Rich Hickey. The clojure programming language. In *Proceedings of the 2008 Symposium on Dynamic Languages*, DLS '08, New York, NY, USA, 2008. ACM.

Marco Emanuel Morais. Comparing spatially explicit models of fire spread through chaparral fuels: A new model based upon the rothermel fire spread equation. Master's thesis, University of California, Santa Barbara, 2001.

Seth H Peterson, Marco E Morais, Jean M Carlson, Philip E Dennison, Dar A Roberts, Max A Moritz, David R Weise, et al. Using hfire for spatial modeling of fire in shrublands. 2009.

Seth H Peterson, Max A Moritz, Marco E Morais, Philip E Dennison, and Jean M Carlson. Modelling long-term fire regimes of southern california shrublands. *International Journal of Wildland Fire*, 20(1): 1–16, 2011.

Richard C Rothermel. A mathematical model for predicting fire spread in wildland fuels. *Research paper/Intermountain forest and range experiment station. USDA (INT-115)*, 1972.

Richard C Rothermel. How to predict the spread and intensity of forest and range fires. *General technical report/Intermountain Forest and Range Experiment Station. USDA (no. INT-143)*, 1983.

Richard C Rothermel. Predicting behavior and size of crown fires in the northern rocky mountains. *Research paper/United States Department of Agriculture, Intermountain Research Station (INT-438)*, 1991.

Joe H Scott and Robert E Burgan. Standard fire behavior fuel models: a comprehensive set for use with rothermel's surface fire spread model. *The Bark Beetles, Fuels, and Fire Bibliography*, page 66, 2005.

Charles E Van Wagner. Conditions for the start and spread of crown fire. *Canadian Journal of Forest Research*, 7(1):23–34, 1977.