The randomwalk package: customizable random walks[*]

Bruno Le Floch[†]

Released on 2018/12/28

Contents

Abstract

The randomwalk package draws random walks. The following parameters can be customized:

- The number of steps, of course.
- The length of the steps, either a fixed length, or a length taken uniformly at random from a given list.
- The angle of each step, either taken uniformly at random from a given list, or uniformly distributed between 0 and 360 degrees.

---

[*] This file describes version v0.6, last revised 2018/12/28.

[†] E-mail blflatex+randomwalk@gmail.com

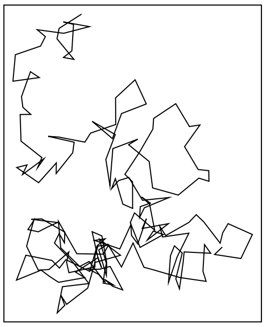

Figure 1: A 200 steps long walk, where each step has one of two lengths: `\RandomWalk {number = 200, length = {4pt, 10pt}}`

1 How to use **randomwalk**

`\RandomWalk`

The **randomwalk** package has a single user command: `\RandomWalk`, which takes a list of key-value pairs as its argument. A few examples are given in Figures 1, 2, and 3:

```
\RandomWalk {number = 200, length = {4pt, 10pt}}
\RandomWalk {number = 100, angles = {0,60,120,180,240,300}, degree}
\RandomWalk {number = 50, length = 1ex, angles = {0,24,48,-24,-48},
degree, angles-relative}
```

Here is a list of all the keys, and their meaning:

- `number`: the number of steps (default 10)
- `length`: the length of each step: either one dimension (*e.g.*, `1ex`), or a comma-separated list of dimensions (*e.g.*, `{2pt, 5pt}`), by default `10pt`. The length of each step is a (uniformly distributed) random element in this set of possible dimensions.
- `angles`: the polar angle for each step: a comma-separated list of angles, and each step takes a random angle in the list. If this is not specified, then the angle is uniformly distributed along the circle.
- `degree` or `degrees`: specify that the angles are given in degrees (by default, they are in radians).
- `angles-relative`: instead of being absolute, the angles are relative to the direction of the previous step.
- `revert-random` (boolean, false by default): revert the seed of the random number generator to its original value after the random walk.

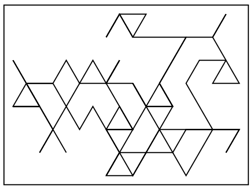

Figure 2: A walk with constrained angles: `\RandomWalk {number = 100, angles = {0,60,120,180,240,300}, degree}`

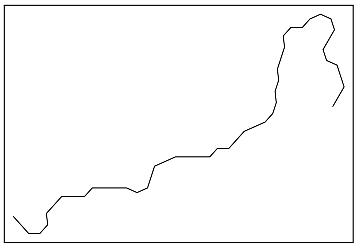

Figure 3: A last example, with small relative angles: `\RandomWalk {number = 50, length = 1ex, angles = {0,24,48,-24,-48}, degree, angles-relative}`

2 randomwalk implementation

2.1 Packages

The expl3 bundle is loaded first.

<*package>

```
⟨@@=randomwalk⟩
```

```
\RequirePackage{expl3}[2017/11/14]
\ProvidesExplPackage
  {randomwalk} {2018/12/28} {0.6} {Customizable random walks}
\RequirePackage{xparse}[2017/11/14]
```

Load pgfcore for figures.

```
\RequirePackage{pgfcore}
```

Load lcg for random numbers. It needs to know the smallest and biggest random numbers that should be produced, which we take to be 0 and `\c__randomwalk_lcg_last_int` = $2^{31}-2$. It will then store them in `\c@lcg@rand`: the `\c@` is there because of how LaTeX 2ε defines counters. To make it clear that `\c` has a very special meaning here, I do not follow LaTeX3 naming conventions. Also of note is that I use `\cr@nd` in `\__randomwalk_walk:`.

It seems that the lcg package has to be loaded after the document class, hence we do it `\AtBeginDocument`. Also worth noting is the call to `\rand`, which avoids some very odd bug.

```
\int_const:Nn \c__randomwalk_lcg_last_int { \c_max_int - 1 }
\AtBeginDocument
  {
    \RequirePackage
      [
        first= 0 ,
        last = \c__randomwalk_lcg_last_int ,
        counter = lcg@rand
      ]
      { lcg }
    \rand
  }
```

2.2 Variables

`\l__randomwalk_internal_tl`
`\l__randomwalk_internal_int`

Used for scratch assignments.

```
\tl_new:N \l__randomwalk_internal_tl
\int_new:N \l__randomwalk_internal_int
```

(*End definition for* `\l__randomwalk_internal_tl` *and* `\l__randomwalk_internal_int`*.*)

`\l__randomwalk_step_number_int`

The number of steps requested by the caller.

```
\int_new:N \l__randomwalk_step_number_int
```

(*End definition for* `\l__randomwalk_step_number_int`*.*)

`\l__randomwalk_relative_angles_bool`
`\l__randomwalk_degrees_bool`

Booleans for whether angles are relative (keyval option), and whether they are in degrees.

```
\bool_new:N \l__randomwalk_relative_angles_bool
\bool_new:N \l__randomwalk_degrees_bool
```

(*End definition for* `\l__randomwalk_relative_angles_bool` *and* `\l__randomwalk_degrees_bool`*.*)

`\l__randomwalk_revert_random_bool`

Booleans for whether to revert the random seed to its original value or keep the last value reached at the end of a random path.

```
\bool_new:N \l__randomwalk_revert_random_bool
```

(*End definition for* `\l__randomwalk_revert_random_bool`*.*)

`\__randomwalk_next_angle:`
`\__randomwalk_next_length:`

Set the `\l__randomwalk_angle_fp` and `\l__randomwalk_length_fp` of the next step, most often randomly.

```
\cs_new_protected:Npn \__randomwalk_next_angle: { }
\cs_new_protected:Npn \__randomwalk_next_length: { }
```

(*End definition for* `\__randomwalk_next_angle:` *and* `\__randomwalk_next_length:`*.*)

`\l__randomwalk_angle_fp`
`\l__randomwalk_length_fp`

Angle and length of the next step.

```
\fp_new:N \l__randomwalk_angle_fp
\fp_new:N \l__randomwalk_length_fp
```

(*End definition for* `\l__randomwalk_angle_fp` *and* `\l__randomwalk_length_fp`*.*)

`\l__randomwalk_x_dim`
`\l__randomwalk_y_dim`

Current coordinates: each `\pgfpathlineto` statement goes from the previous value of these to the next. See `\__randomwalk_walk_step:`.

```
\dim_new:N \l__randomwalk_x_dim
\dim_new:N \l__randomwalk_y_dim
```

(*End definition for* `\l__randomwalk_x_dim` *and* `\l__randomwalk_y_dim`*.*)

`\l__randomwalk_angles_seq`
`\l__randomwalk_lengths_seq`

Sequences containing all allowed angles and lengths, as floating point numbers.

```
\seq_new:N \l__randomwalk_angles_seq
\seq_new:N \l__randomwalk_lengths_seq
```

(*End definition for* `\l__randomwalk_angles_seq` *and* `\l__randomwalk_lengths_seq`*.*)

2.3 User command and key-value list

`\RandomWalk`
`\randomwalk:n`

The user command `\RandomWalk` is based on the code-level command `\randomwalk:n`, which simply does the setup and calls the internal macro `\__randomwalk_walk:`.

```
\DeclareDocumentCommand \RandomWalk { m }
  { \randomwalk:n {#1} }
\cs_new_protected:Npn \randomwalk:n #1
  {
    \__randomwalk_setup_defaults:
    \keys_set:nn { randomwalk } {#1}
    \__randomwalk_walk:
  }
```

(*End definition for* `\RandomWalk` *and* `\randomwalk:n`*. These functions are documented on page 2.*)

We introduce the keys for the package.

```
\keys_define:nn { randomwalk }
  {
    number .value_required:n = true ,
    length .value_required:n = true ,
    angles .value_required:n = true ,
    number .int_set:N = \l__randomwalk_step_number_int ,
    length .code:n = { \__randomwalk_setup_length:n {#1} } ,
    angles .code:n = { \__randomwalk_setup_angles:n {#1} } ,
    degree .bool_set:N = \l__randomwalk_degrees_bool ,
    degrees .bool_set:N = \l__randomwalk_degrees_bool ,
    angles-relative .bool_set:N = \l__randomwalk_relative_angles_bool ,
    revert-random .bool_set:N = \l__randomwalk_revert_random_bool ,
  }
```

2.4 Setup

`\__randomwalk_setup_defaults:`

The package treats the length of steps, and the angle, completely independently. The function `\__randomwalk_next_length:` contains the action that decides the length of the next step, while the function `\__randomwalk_next_angle:` pertains to the angle.

`\__randomwalk_setup_defaults:` sets the default values before processing the user's key-value input. This also sets initial values of variables that currently cannot be altered through keys, because it might be good to provide keys for their initial values too later on.

```
\cs_new_protected:Npn \__randomwalk_setup_defaults:
  {
    \int_set:Nn \l__randomwalk_step_number_int {10}
    \cs_gset_protected:Npn \__randomwalk_next_angle:
      { \__randomwalk_fp_set_rand:Nnn \l__randomwalk_angle_fp { 0 } { 360 } }
    \cs_gset_protected:Npn \__randomwalk_next_length:
      { \fp_set:Nn \l__randomwalk_length_fp {10} }
    \bool_set_false:N \l__randomwalk_revert_random_bool
    \bool_set_false:N \l__randomwalk_relative_angles_bool
    \fp_zero:N \l__randomwalk_angle_fp
    \fp_zero:N \l__randomwalk_length_fp
    \dim_zero:N \l__randomwalk_x_dim
    \dim_zero:N \l__randomwalk_y_dim
  }
```

(*End definition for* `\__randomwalk_setup_defaults:`*.*)

`\__randomwalk_setup_length:n` Convert each item in the comma list into a floating point, then define `\__randomwalk_next_length:` to set `\l__randomwalk_length_fp` to a random floating point in the list.

```
\cs_new_protected:Npn \__randomwalk_setup_length:n #1
  {
    \seq_set_split:Nnn \l__randomwalk_lengths_seq { , } {#1}
    \seq_set_map:NNn \l__randomwalk_lengths_seq
      \l__randomwalk_lengths_seq { \dim_to_fp:n {##1} }
    \cs_gset_protected:Npn \__randomwalk_next_length:
      {
        \__randomwalk_get_rand_seq_item:NN
          \l__randomwalk_lengths_seq \l__randomwalk_internal_tl
        \fp_set:Nn \l__randomwalk_length_fp { \l__randomwalk_internal_tl }
      }
  }
```

(*End definition for* `\__randomwalk_setup_length:n`*.*)

`\__randomwalk_setup_angles:n` Two complications compared to `\__randomwalk_setup_length:n`. First, the angle can be given in radians rather than degrees: then add `rad` after the randomly chosen value (in principle it would be better to convert angles once and for all at the beginning, but that interacts in a complicated way with the fact that keys can be given in any order). Second, angles can be relative, in which case we use `\fp_add:Nn` to take the last angle into account.

```
\cs_new_protected:Npn \__randomwalk_setup_angles:n #1
  {
    \seq_set_split:Nnn \l__randomwalk_angles_seq { , } {#1}
    \seq_set_map:NNn \l__randomwalk_angles_seq
      \l__randomwalk_angles_seq { \fp_to_tl:n {##1} }
    \cs_gset_protected:Npn \__randomwalk_next_angle:
      {
        \__randomwalk_get_rand_seq_item:NN
          \l__randomwalk_angles_seq \l__randomwalk_internal_tl
        \bool_if:NF \l__randomwalk_degrees_bool
          { \tl_put_right:Nn \l__randomwalk_internal_tl { rad } }
        \bool_if:NTF \l__randomwalk_relative_angles_bool
          { \fp_add:Nn } { \fp_set:Nn }
          \l__randomwalk_angle_fp { \l__randomwalk_internal_tl }
      }
  }
```

(*End definition for* `\__randomwalk_setup_angles:n`*.*)

2.5 Drawing

`\__randomwalk_walk:` We are ready to define `\__randomwalk_walk:`, which draws a `pgf` picture of a random walk with the parameters set up by the `keys`. We reset coordinates to zero originally. Then draw the relevant `pgf` picture by repeatedly calling `\__randomwalk_walk_step:`.

```
\cs_new_protected:Npn \__randomwalk_walk:
  {
    \__randomwalk_walk_start:
    \prg_replicate:nn { \l__randomwalk_step_number_int }
```

```
      { \__randomwalk_walk_step: }
    \bool_if:NF \l__randomwalk_revert_random_bool
      { \int_gset_eq:NN \cr@nd \cr@nd }
    \__randomwalk_walk_stop:
  }
```

`\cr@nd` is internal to the lcg package.

(*End definition for* `\__randomwalk_walk:`*.*)

`\__randomwalk_walk_start:`
`\__randomwalk_walk_line:`
`\__randomwalk_walk_stop:`

These functions encapsulate all of the `pgf`-related code. The `start` function begins the pgfpicture environment and starts a path at position (x,y). The `line` function adds to the path a line from the previous position to the new (x,y). The `stop` function draws the path constructed by `\__randomwalk_walk_step:` and ends the pgfpicture environment.

```
\cs_new_protected:Npn \__randomwalk_walk_start:
  {
    \begin{pgfpicture}
      \pgfpathmoveto
        { \pgfpoint { \l__randomwalk_x_dim } { \l__randomwalk_y_dim } }
  }
\cs_new_protected:Npn \__randomwalk_walk_line:
  {
    \pgfpathlineto
      { \pgfpoint { \l__randomwalk_x_dim } { \l__randomwalk_y_dim } }
  }
\cs_new_protected:Npn \__randomwalk_walk_stop:
  {
      \pgfusepath { stroke }
    \end{pgfpicture}
  }
```

(*End definition for* `\__randomwalk_walk_start:`*,* `\__randomwalk_walk_line:`*, and* `\__randomwalk_walk_stop:`*.*)

`\__randomwalk_walk_step:`

`\__randomwalk_walk_step:` calls `\__randomwalk_next_length:` and `\__randomwalk_next_angle:` to determine the length and angle of the new step. This is then converted to cartesian coordinates and added to the previous end-point. Finally, we call `pgf`'s `\pgfpathlineto` to produce a line to the new point.

```
\cs_new_protected:Npn \__randomwalk_walk_step:
  {
    \__randomwalk_next_length:
    \__randomwalk_next_angle:
    \dim_add:Nn \l__randomwalk_x_dim
      {
        \fp_to_dim:n
          { \l__randomwalk_length_fp * cosd ( \l__randomwalk_angle_fp ) }
      }
    \dim_add:Nn \l__randomwalk_y_dim
      {
        \fp_to_dim:n
          { \l__randomwalk_length_fp * sind ( \l__randomwalk_angle_fp ) }
      }
    \__randomwalk_walk_line:
  }
```

(*End definition for* `\__randomwalk_walk_step:`*.*)

2.6 On random numbers and items

For random numbers, the interface of lcg is not quite enough, so we provide our own LaTeX3-y functions. Also, this will allow us to change quite easily our source of random numbers.

`\__randomwalk_fp_set_rand:Nnn`

We also need floating point random numbers, assigned to the variable `#1`.

```
\cs_new_protected:Npn \__randomwalk_fp_set_rand:Nnn #1#2#3
  {
    \rand
    \fp_set:Nn #1
      { #2 + (#3 - (#2)) * \c@lcg@rand / \c__randomwalk_lcg_last_int }
  }
```

(*End definition for* `\__randomwalk_fp_set_rand:Nnn`.)

`\__randomwalk_get_rand_seq_item:NN`

We can now pick an element at random from a sequence. If the sequence has a single element, no need for randomness.

```
\cs_new_protected:Npn \__randomwalk_get_rand_seq_item:NN #1#2
  {
    \int_set:Nn \l__randomwalk_internal_int { \seq_count:N #1 }
    \int_compare:nTF { \l__randomwalk_internal_int = 1 }
      { \tl_set:Nx #2 { \seq_item:Nn #1 { 1 } } }
      {
        \rand
        \tl_set:Nx #2
          {
            \seq_item:Nn #1
              {
                1 +
                \int_mod:nn { \c@lcg@rand } { \l__randomwalk_internal_int }
              }
          }
      }
  }
```

(*End definition for* `\__randomwalk_get_rand_seq_item:NN`.)

```
⟨/package⟩
```