Stefano Maggiolo

The Time It Takes To Change The Time

HACKERMONTHLY Issue 66 November 2015

Curator
Lim Cheng Soon

Contributors
Stefano Maggiolo
Rodrigo Monteiro
Tim Babb
Jeff Bradberry
Ben Einstein
Ian Murdock

Proofreader
Emily Griffin

Printer
Blurb

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format.
For more, visit *hackermonthly.com*

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.

Cover Illustration: Dennis [catch---22.deviantart.com]

Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-66*

The Time It Takes To Change The Time

By STEFANO MAGGIOLO

You may remember my map showing the difference between solar time and standard time from last year, as it was by far the most shared content I created. In fact, somebody even uploaded it to Wikipedia for the time zone article! But with great power comes great responsibility (…); and in this case, that means keeping the map up-to-date as time regulation evolves. You may think that this is a rare event, but when you consider the 200 or so sovereign states in the world, and the quadratic number of possible conflicts between them, it occurs quite frequently.

Indeed, the triggering event that prompted me to draw a new version of the old map wasn't the poor color choice of the first attempt (for many people, green and red do not play well together), nor did it have anything to do with the minor mistakes in the zone divisions. Moreover, my new map was completely unrelated to any political faux pas (for example, not marking Taiwan in the same way as most other sovereign states). After all, the map was drawn on top of a preexisting one on Wikipedia, with only minor modifications made (apart from the gradients, that is).

The real reason behind the new map is that just a few months after releasing the original version, Russia decided to change the time in most of the country. Since Russia extends to about 3.5% of the world's area, and 11.5% of the emerged lands area, the issue was too substantial to ignore.

Aside from this big change, the new map reflects that certain territories in Ukraine and Georgia follow Moscow time instead of their countries' own timezone. Further, it accommodates the introduction of the "Southeast" timezone (permanent UTC-05:00) in Quintana Roo, Mexico. In terms of longitude, Continental Mexico is about 30 degrees wide, from Tijuana to Cancun, so two timezones would make sense. However, the nation has four!

Solar time vs standard time

Overall, the map is still skewed toward the red (meaning that the solar noon occurs later in the day), and most of what could be observed from the first version is still valid. In particular, for the joy of reddit commentators, China is still very red.

On the other hand, one change wasn't implemented: Australia still refuses to make Central Western Time — aka the awesome UTC+08:45 timezone — official. You see how the difference between Western and Central Australia is a whole hour and a half? The problem isn't too significant because no people live at the transition zone, apart from a very narrow strip of 350km along the southern coast, on the road between Perth and Adelaide, where about 200 proud residents literally live on their own time.

If made official, it would be the timezone with the fewest residents (followed by UTC+10:30, Lord Howe Island, with about 360 inhabitants, and UTC+12:45, Chatham Islands, with roughly 600), apart from the ephemeral UTC-12:00 timezone: Only a few birds are known to inhabit the two US islands of Howland and Baker, with their whopping 4 squared kilometers. However, since the territories are uninhabited, nobody ever determined a timezone for these two islands, and they are unofficially UTC-12:00 — just because they happen to be at the correct longitude!

I mentioned that 90 minutes is an unusually large difference between neighboring timezones, since one hour is the standard. That may be the case, but there are still many places with much larger differences (and these places often involve China, thanks to the nation's single, large-scale timezone). The worst offender is the 90-kilometer border between Afghanistan (UTC+04:30) and China (UTC+08:00), which spans a difference of three and a half hours!

How to draw a map

Drawing the first version of this map was a very long process, done more or less by hand. I spent quite a bit of time modifying the source of an existing SVG file, and I most certainly did not want to go through the same process again this time around. As such, I blatantly ignored xkcd and proceeded to Create the Tool that would create the map for me. For the future, the real value of the tool depends on the availability of up-to-date data — still, I can trick myself into thinking I've saved time.

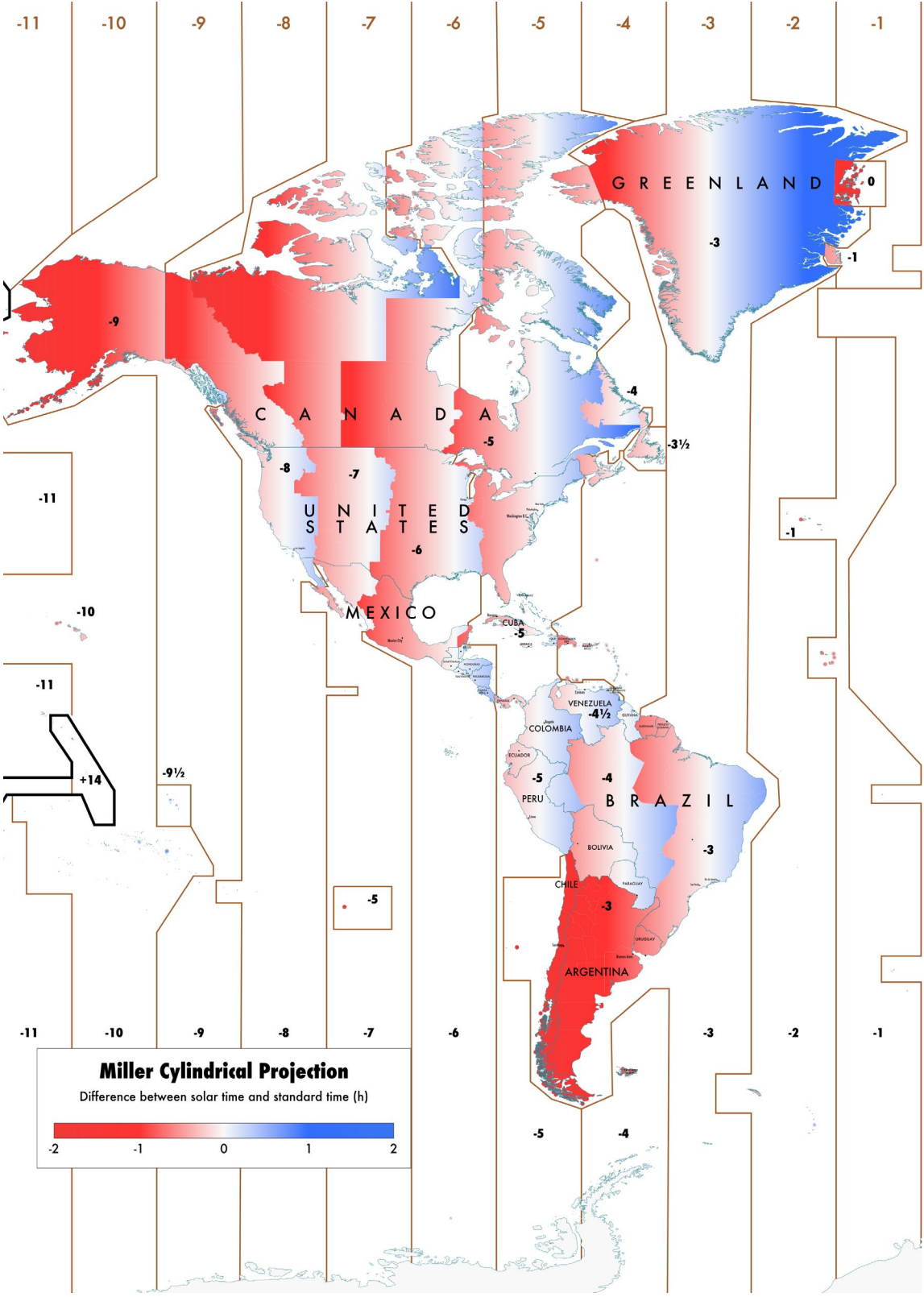
All the tools mentioned below can be found in the solar time vs standard time repository. When people look at maps, they often fail to realize how many ingredients combine to create them; we take a lot of things for granted. This map probably has a below-average number of ingredients, although we can see:

- timezone boundaries over the sea (approximate, as the territorial water boundaries are too intricate to draw);
- land/water and country boundaries;
- timezone territories (which are a different hierarchy than countries; a country may have more than one timezone, and a timezone may encompass multiple countries);
- city coordinates;
- labels of:
  - countries;
  - cities;
  - timezones.

Fortunately, there are datasets available for most of these; some with a unique source — maybe slightly out-of-date — and some with more sources. Choosing the best options in terms of accuracy and license is an important step when creating a map programmatically. In our case, timezones come from Eric Muller's website, and the same goes for national boundaries (these, in turn, come from the FIPS dataset). Homogeneity, furthermore, is an important quality in a good map, and since timezone boundaries are a very rare dataset, it made sense to take the national boundaries from Muller as well. Conversely, the coordinates of major cities (including other useful informational to select whose to show) were taken from the ESRI datasets.

Getting the data, however, is just the first step. The second isn't too difficult, though: Draw the SVG with the boundaries. This involves some templating to create the file, and most importantly, translating latitude and longitude into pixel coordinates (that is, the choice of a projection). Again, I was constrained to a rectilinear projection, as this made drawing the gradients was much simpler, so I stuck with the Muller information.

The third step of the map-making process is writing the labels — and this involves an important choice: You must determine whether it's faster to design an algorithm that tries to place the labels automatically, or to place them semi-manually. The first option is tough; I would frame it as an optimization problem, where the function to optimize depends on the distance between the labels and that to which they refer — on their size, and on how much they overlap — but I'll admit I did not go for this option. Instead, the semi-manual approach consists of placing the labels in a reasonable position (at the centroid of the country, and anchored to the coordinates of the city), and then to tweak whatever labels might need tweaking.



Then, the fourth step is to draw the labels and lines of the timezones. For the labels, I wrote a list of positions I wanted — so again, the process was very manual. For the lines, in theory, a good approximation is to draw 24 equispaced meridians, but then the map becomes very hard to read. To make the job of drawing multiple polylines on the map, I wrote a very simple helper tool based on Maps API, which lets the user draw directly on the map and retrieve the coordinates of the vertices in JSON format.

Finally, the last step: data correction! Unfortunately, Eric Muller's data is not completely up-to-date. Some fixes were easy (just changing the offset of a few timezones), while others weren't quite as simple, and I eventually decided to use GIMP to draw over the final

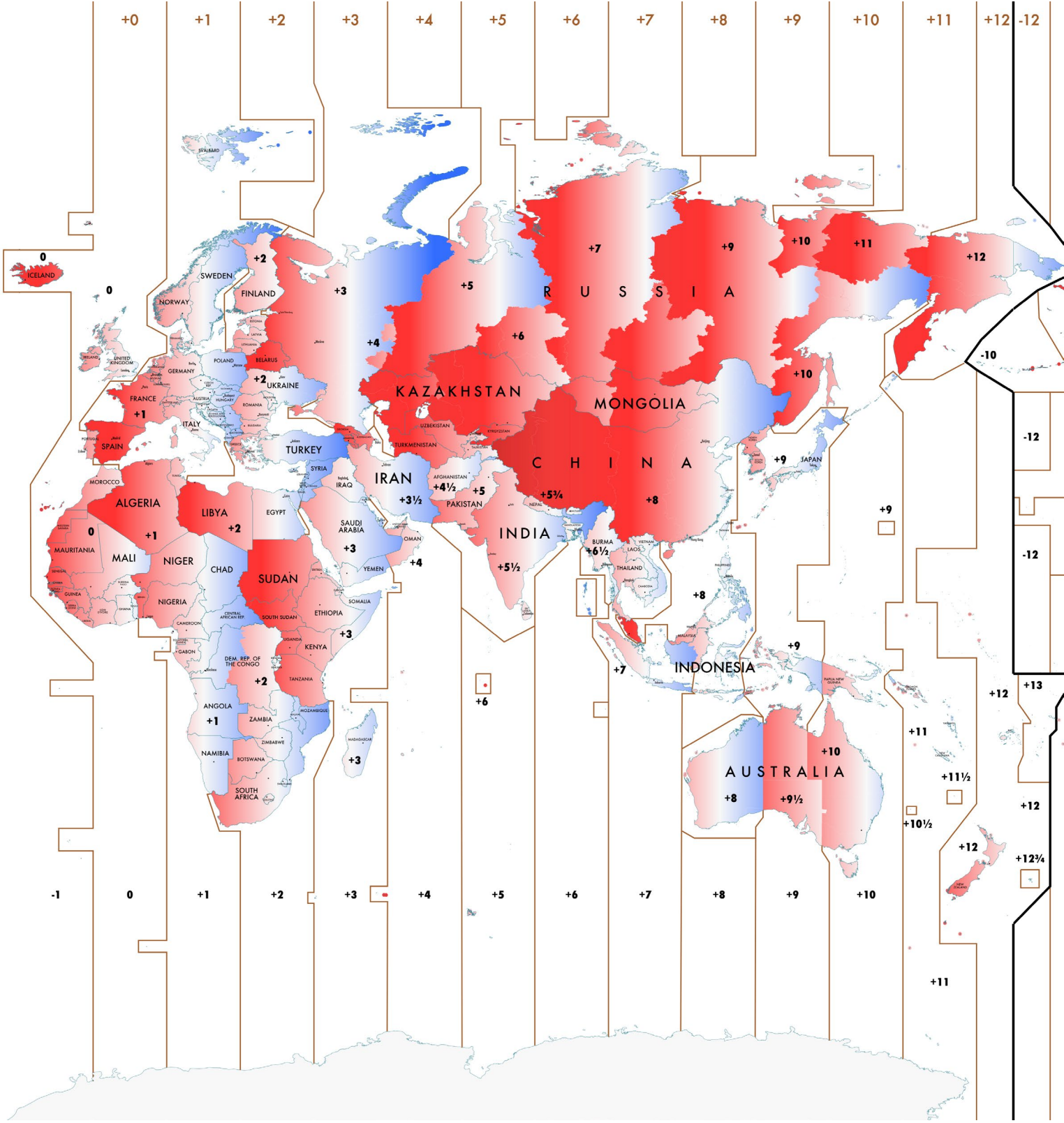


image generated by the program (This explains the GIMP file in the repository.) In post-processing, I also shifted the center of the map so that the cut point was not over Siberia; in theory, it should have been easy enough to do in the projection stage, but the source datasets already split all lands on the 180 meridian, which made it more convenient to use the same cut point in the projection. ■

Stefano has a background in programming competitions, Mathematics, and Geometry. Being true to the root of the word "geometry", he loves maps and works as a software engineer in Google's location team in London.

Reprinted with permission of the original author.
First appeared in *hn.my/time* (poormansmath.net)

The Guide To Implementing 2D Platformers

By RODRIGO MONTEIRO

HAVING PREVIOUSLY BEEN disappointed by the information available on the topic, this is my attempt at categorizing different ways to implement 2D platform games, list their strengths and weaknesses, and discuss some implementation details.

The long-term goal is to make this an exhaustive and comprehensible guide to the implementation of 2D platform games. If you have any sort of feedback, correction, request or addition — please leave it in the comments!

Disclaimer: some of the information presented here comes from reverse engineering the behavior of the game, not from its code or programmers. It's possible that they are not ACTUALLY implemented in this way, and merely behave in an equivalent way. Also, note that tile sizes are for the game logic, graphical tiles might be of a different size.

FOUR WAYS OF IMPLEMENTING

I can think of four major ways in which a platform game can be implemented. From simplest to most complicated, they are:

Type #1: Tile-based (pure)

Character movement is limited to tiles, so you can never stand halfway between two tiles. Animations may be used to create the illusion of smooth movement, but as far as the game logic is concerned, the player is always right on top of a specific tile. This is the easiest way to implement a platform game, but it imposes heavy restrictions on the control of the character, making it unsuitable for traditional action-based platformers. It is, however, popular with puzzle and "cinematographic" platformers.

Flashback, shown with tile boundaries

Examples: Prince of Persia, Toki Tori, Lode Runner, Flashback

How it works

The map is a grid of tiles, each one storing information such as whether it's an obstacle or not, what image to use, what kind of footstep sound to use, and so on. The player and other characters are represented by a set of one or more tiles that move together. In Lode Runner, for example, the player is a single tile. In Toki Tori, the player is 2×2 tiles. In Flashback, which is unusual due to the smaller size of its tiles, the player is two tiles wide and five tiles tall (see image above) when standing, but they are only three tiles tall when crouching.

In this kind of game, the player will rarely — if ever — be moving diagonally, but, if he is, the movement can be decomposed in two separate steps. Likewise, he will likely only move one tile at once, but multi-tile movement can be done as multiple steps of one tile if needed (in Flashback, you always move two tiles at once). The algorithm is then as follows:

1. Create a copy of the character where he'd like to move to (e.g., if moving one tile to the right, make a copy where every tile of the character is shifted 1 tile to the right)
2. Check that copy for intersection with the background and other characters.
3. If an intersection is found, the character's movement is blocked. React accordingly.
4. Otherwise, the path is clear. Move character there, optionally playing an animation so the transition looks smooth.

This kind of movement is very ill-suited for traditional arc-shaped jumps — so games in this genre often have no jump at all (Toki Tori, Lode Runner), or only allow vertical or horizontal jumps (Prince of Persia, Flashback), which are nothing but special cases of linear movement.

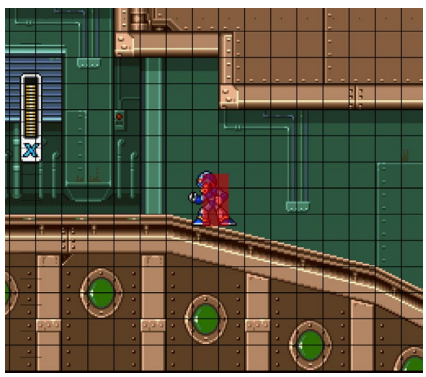
Advantages of this system include simplicity and precision. Since the games are more deterministic, glitches are much less likely and the gameplay experience is more controlled with less of a need to tweak values depending on circumstances. Implementing certain mechanics (such as grabbing ledges and one-way platforms) becomes a breeze, compared to more complex movement styles — all you have to do is check whether the player tiles and the background tiles are aligned in the one specific way that allows for a given action.

In principle, this system doesn't allow steps of less than one tile, but that can be mitigated in a few different ways. For example, the tiles can be a bit smaller than the player (say, a player is 2×6 tiles), or you can allow a visual-only movement to take place inside a given tile, without affecting the logic (which is the solution that I believe that "Lode Runner — The Legend Returns" takes).

Type #2: Tile Based (Smooth)

Collision is still determined by a tilemap, but characters can move freely around the world (typically with 1px resolution, aligned to integers, but see the note at the end of the article regarding smoothing of movement). This is the most common form of implementing platformers in 8-bit and 16-bit consoles, and remains popular today, as it is still easy to implement and makes level editing simpler than more sophisticated techniques. It also allows for slopes and smooth jump arcs.

If you're unsure which type of platformer you want to implement, and you want to do an action game, I suggest going for this one. It's very flexible, relatively easy to implement, and gives you the most control of all four types. It's no wonder that the majority of the best action platformers of all time are based on this type.



Mega Man X, shown with tile boundaries and player hitbox.

Examples: Super Mario World, Sonic the Hedgehog, Mega Man, Super Metroid, Contra, Metal Slug, and practically every platformer of the 16-bit era

How it works

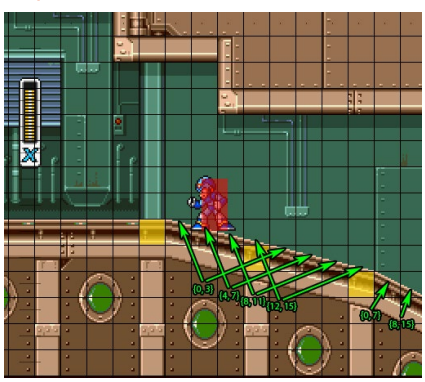
Map information is stored in the same way as with the pure tile technique, the difference is merely in how the characters interact with the background. The character's collision hitbox is now an Axis-Aligned Bounding Box (AABB, that is, a rectangle that cannot be rotated), and are typically still an integer multiple of tile size. Common sizes include one tile wide and one (small Mario, morph ball Samus), two (big Mario, Mega Man, crouched Samus) or three (standing Samus) tiles tall. In many cases, the character sprite itself is larger than the logical hitbox, as this makes for a more pleasant visual experience and fairer gameplay (it's better for the player to avoid being hit when he should have than for him to get hit when he should not have). In the image above, you can see that the sprite for X is square-ish (in fact, it is two tiles wide), but his hitbox is rectangular (one tile wide).

Assuming that there are no slopes and only one-way platforms, the algorithm is straightforward:

1. Decompose movement into X and Y axes, step one at a time. If you're planning on implementing slopes afterward, step X first, then Y. Otherwise, the order shouldn't matter much. Then, for each axis:
2. Get the coordinate of the forward-facing edge, e.g., if walking left, the x coordinate left of the bounding box. If walking right, x coordinate of the right side. If up, y coordinate of the top, etc.

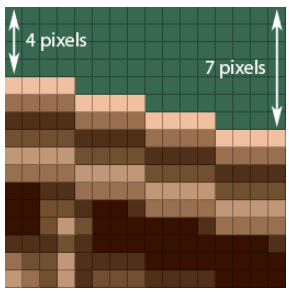
3. Figure out which lines the tiles of the bounding box intersect with — this will give you a minimum and maximum tile value on the OPPOSITE axis. For example, if we're walking left, perhaps the player intersects with horizontal rows 32, 33 and 34 (that is, tiles with y = 32 * TS, y = 33 * TS, and y = 34 * TS, where TS = tile size).
4. Scan along those lines of tiles and toward the direction of movement until you find the closest static obstacle. Then loop through every moving obstacle, and determine which of these obstacles is the closest and actually on your path.
5. The total movement of the player along that direction is then the minimum between the distance to the closest obstacle and the amount that you wanted to move in the first place.
6. Move player to the new position. With this new position, step the other coordinate, if still not done.

Slopes



Mega Man X, with slope tile annotations

Slopes (the tiles pointed out by green arrows on the image above) can be very tricky because they are obstacles, and yet still allow the character to move onto their tile. They also cause movement along the x-axis to adjust position on the y-axis. One way to deal with them is to have the tile store the "floor y" on either side. Assuming a coordinate system where (0, 0) is at top-left, then the tile just left of X (first slope tile) is {0, 3} (left, right), then the one he stands on is {4, 7}, then {8, 11}, then {12, 15}. After that, the tiles repeat, with another {0, 3}, etc., and then we have a steeper slope, composed of two tiles: {0, 7} and {8, 15}.



Detailed View of the {4, 7} tile

The system that I'm going to describe allows arbitrary slopes, although, for visual reasons, those two slopes are the most common, and result in a total of 12 tiles (the 6 described previously, and their mirrorings). The collision algorithm changes as follows for horizontal movement:

- Make sure that you step X position before Y position.
- During collision detection (4 above), the slope only counts as a collision if its closest edge is the taller (smaller y coordinate) one. This will prevent characters from "popping" through the slope from the opposite side.
- You might want to forbid slopes to stop "halfway through" (e.g., on a {4, 7} tile). This restriction is adopted by Mega Man X and many other games. If you don't, you have to deal with the more complicated case of the player attempting to climb from the lower side of the slope tile — one way to deal with this is to pre-process the level, and flag all such offending tiles. Then, on collision detection, also count it as a collision from the lower side if the player's lowest y coordinate is greater (that is, below) the tile's offset edge (tile coordinates * tile size + floor y).
- A full obstacle tile adjacent to the slope the character is currently on should not be considered for collision if it connects to the slope, that is, if the character (that is, his bottom-center pixel) is on a {0, *} slope, then ignore left tile, and, if on a {*, 0} slope, then ignore the right tile. You may have to do this for more tiles if your character is wider than two tiles — you might simply skip checking on the entire row if the player is moving toward the upper side of the slope. The reason for this is to prevent the character from getting stuck at those tiles (highlighted yellow above) while still climbing the slope, as his foot will still be below the "surface level" by the time he comes into contact with the otherwise solid tile.

And for vertical movement:

- If you're letting gravity do its job for downhill movement, make sure that the minimum gravity displacement is compatible with slope and horizontal velocity. For example, on a 4:1 slope (as {4, 7} above), the gravity displacement must be at least 1/4 of the horizontal velocity, rounded up.

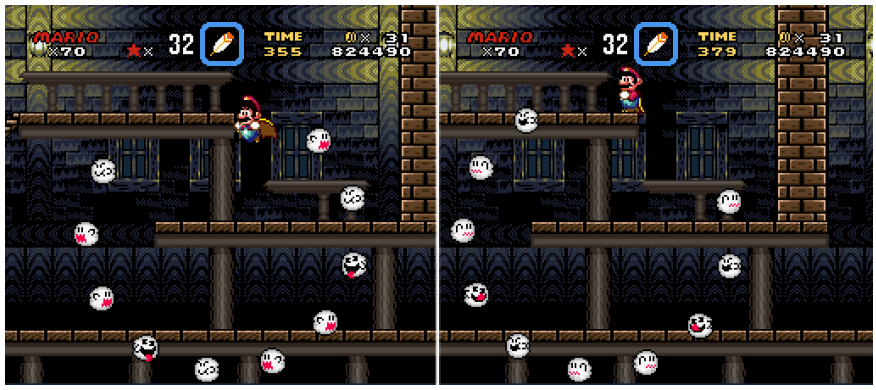
On a 2:1 slope (such as {0, 7}), at least 1/2. If you don't ensure this, the player will move horizontally right off the ramp for a while, until gravity catches up and drags him down, making him bounce on the ramp, instead of smoothly descending it.

- An alternative to using gravity is to compute how many pixels above the floor the player was before movement, and how many it is afterward (using the formula below), and adjust his position so they're the same.
- When moving down, instead of considering a slope tile's top edge as its collision boundary; instead, compute its floor coordinate at the current vertical line, and use that. To do that, find the [0, 1] value that represents the player's x position on tile (0 = left, 1 = right) and use it to linearly interpolate the floorY values. The code will look something like:

```
float t = float(centerX - tileX)
/ tileSize;
float floorY = (1-t) * leftFloorY
+ t * rightFloorY;
```

- When moving down, if multiple tiles on the same Y coordinate are obstacle candidates, and the one on the X coordinate of the player's center is a slope tile, use that one and ignore the rest — even though the others are technically closer. This ensures proper behavior around the edges of slopes, with the character actually "sinking" on a completely solid tile because of the adjacent slope.

One-way platforms



Super Mario World - showing Mario falling through (left) and standing on (right) the same one-way platform

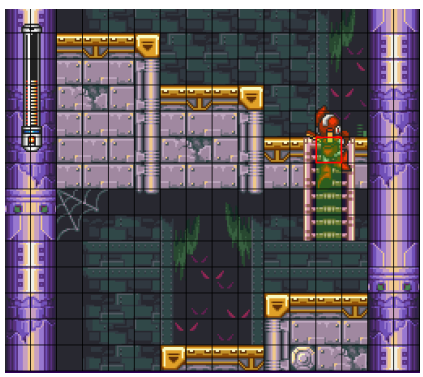
One-way platforms are platforms that you can step on, but can also be jumped through. In other words, they count as an obstacle if you're already on top of them, but are otherwise traversable. That sentence is the key to understanding their behavior. The algorithm changes as follows:

- On the x-axis, the tile is never an obstacle
- On the y-axis, the tile is only an obstacle if, prior to the movement, the player was entirely above it (that is, the bottom-most coordinate of player was at least one pixel above the top-most coordinate of one-way platform). To check for this, you will probably want to store the original player position before doing any stepping.

It might be tempting to have it act as an obstacle if the player's y speed is positive (that is, if the player is falling), but this behavior is wrong: it's possible for the player to jump so he overlaps the platform, but then fall down again without having his feet reach the platform. In that case, he should still fall through.

Some games allow the player to "jump down" from such platforms. There are a few ways to do this, but they are all relatively simple. You could, for example, disable one-way platforms for a single frame and ensure that y speed is at least one (so he'll be clear of the initial collision condition on the next frame), or you could check if he's standing exclusively on one-way platforms, and, if so, manually move the player one pixel down to the bottom.

Ladders



Mega Man 7, with tile boundaries, highlighted ladder tiles, and player ladder hitbox.

Ladders might seem complicated to implement, but they are simply an alternate state — when you're on a ladder, you ignore most of the standard collision system, and replace it

with a new set of rules. Ladders are typically one tile wide.

You can usually enter the ladder state in two ways:

- Have your character hitbox overlap with the ladder, either on ground or on air, and hit up (some games also allow you to hit down)
- Have your character stand on top of a "ladder top" tile (which is often a one-way platform tile as well so you can walk on top of it), and hit down.

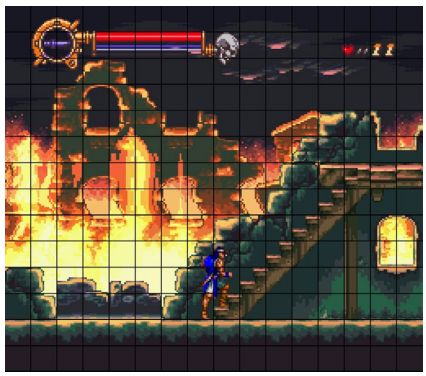
This has the effect of immediately snapping the player's x coordinate to align with the ladder tiles, and, if going down from the top of the ladder, move y coordinate so the player is now inside the actual ladder. At this point, some games will use a different hitbox for the purposes of determining whether the player is still on the ladder. Mega Man, for example, seems to use a single tile (equivalent to the top tile of the original character, highlighted in red in the image above).

There are a few different ways of LEAVING the ladder:

- Reaching the top of the ladder. This will usually prompt an animation and move the player several pixels up in y, so he's now standing on top of the ladder.
- Reaching the bottom of a hanging ladder. This will cause the player to simply fall, although some games won't let the player leave the ladder in this way.
- Moving left or right. If there is no obstacle on that side, the player may be allowed to leave that way.
- Jumping. Some games allow you to release the ladder by doing this.

While on the ladder, the character's movement changes so, typically, all he can do is move up or down and sometimes attack.

Stairs



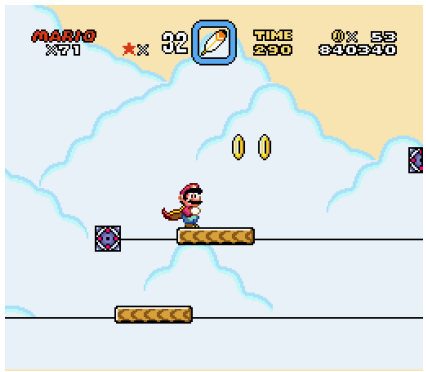
Castlevania - Dracula X, with tile boundaries

Stairs are a variation of ladders, seen in a few games, but notably in the Castlevania series. The actual implementation is very similar to that of ladders, with a few exceptions:

- The player moves tile by tile or half-tile by half-tile (as in Dracula X)
- Each "step" causes the player to be shifted simultaneously on X and Y coordinates, by a preset amount.
- Initial overlapping detection when going up might look on the tile ahead instead of just the current overlapped one.

Other games also have stairs that behave like slopes. In that case, they are simply a visual feature.

Moving Platforms



Super Mario World

Moving platforms can seem a little tricky, but are actually fairly simple. Unlike normal platforms, they cannot be represented by fixed tiles (for obvious reasons), and instead should be represented by an AABB, that is, a rectangle that cannot be rotated. It is a normal obstacle for all collision purposes, and if you stop here, you'll have very slippery moving platforms (that is, they work as intended, except that the character does not move along it on his own).

There are a few different ways to implement that. One algorithm is as follows:

- Before anything on the scene is stepped, determine whether the character is standing on a moving platform. This can be done by checking, for example, whether his center-bottom pixel is just one pixel above the surface of the platform. If it is, store a handle to the platform and its current position inside the character.
- Step all moving platforms. Make sure that this happens before you step characters.

- For every character that's standing on a moving platform, figure the delta-position of the platform, that is, how much it has moved along each axis. Now, shift the character by the same amount.
- Step the characters as usual.

Other Features

Sonic the Hedgehog 2

Other games have more complicated and exclusive features. Sonic the Hedgehog series is notable for this. Those are beyond the scope of this article (and my knowledge, for that matter!), but might be the subject of a future article.

Type #3: Bitmask

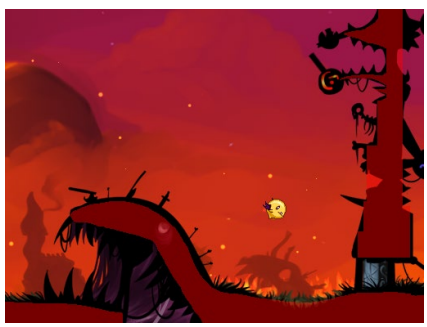
Bitmask is similar to "Tile Based (Smooth)," but instead of using large tiles, an image is used to determine collision for each pixel. This allows finer detailing, but significantly increases complexity, memory usage, and requires something akin to an image editor to create levels. It also often implies that tiles won't be used for visuals, and may, therefore, require large, individual artwork for each level. Due to those issues, this is a relatively uncommon technique but can produce higher quality results than tile-based approaches. It is also suitable for dynamic environments — such as the destructible scenarios in Worms — as you can "draw" into the bitmask to change the scenario.

Worms World Party, featuring destructible terrain

Examples: Worms, Talbot's Odyssey

How it works

The basic idea is very similar to the tile (smooth) algorithm — you can simply consider each pixel to be a tile, implement the exact same algorithm, and everything will work, with one major exception — slopes. Since slopes are now implicitly defined by the positioning between nearby tiles, the previous technique doesn't work, and a much more complex algorithm has to be used in its place. Other things, such as ladders, also become trickier.



Talbot's Odyssey, with the collision bitmask overlaid on top of the game.

Slopes

Slopes are the primary reason why this type of implementation is very hard to get right. Unfortunately, they are also pretty much mandatory, as it'd make no sense to use this implementation without slopes. Often, they're the reason why you're even using this system.

This is, roughly, the algorithm used by Talbot's Odyssey:

- Integrate acceleration and velocity to compute the desired delta-position vector (how much to move on each axis).
- Step each axis separately, starting with the one with the largest absolute difference.
- For the horizontal movement, offset the player AABB by 3 pixels to the top so he can climb slopes.
- Scan ahead, by checking against all valid obstacles and the bitmask itself, to determine how many pixels the character is able to move before hitting an obstacle. Move to this new position.
- If this was horizontal movement, move as many pixels up as necessary (which should be up to 3) to make up for slope.
- If, at the end of the movement, any pixel of the character is overlapping with any obstacle, undo the movement on this axis.
- Regardless of the result of the last condition, proceed to do the same for the other axis.

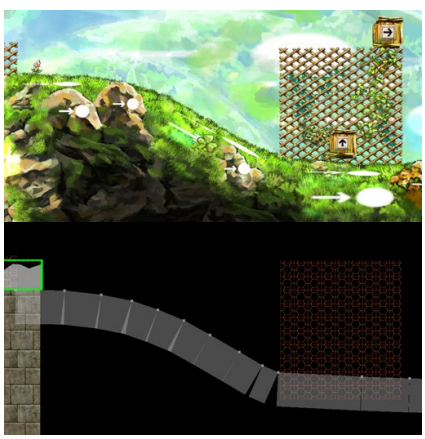
Because this system has no distinction between moving down, because you're going downhill or because you're falling, you're likely to need a system counting how many frames it's been since the character last touched the floor, for purposes of determining whether it can jump and changing animation. For Talbot, this value is 10 frames.

Another trick here is efficiently computing how many pixels the character can move before hitting something. There are other possible complicating factors, such

as one-way platforms (dealt in the exact same way as for tiled (smooth)) and sliding down steep inclines (which is fairly complex and beyond the scope of the article). In general, this technique requires a lot of fine-tuning and is intrinsically less stable than tile-based approaches. I only recommend it if you absolutely must have detailed terrain.

Type #4: Vectorial

This technique uses vectorial data (lines or polygons) to determine the boundaries of collision areas. Very difficult to implement properly, it is nevertheless increasingly popular due to the ubiquity of physics engines, such as Box2D, which are suitable for implementing this technique. It provides benefits similar to the bitmask technique, but without major memory overhead, using a very different method of editing levels.



Braid (level editor), with visible layers (top) and the collision polygons (bottom)

Examples: Braid, Limbo

How it works

There are two general ways of approaching this:

- Resolve movement and collisions yourself, similar to the bitmask method, but using polygon angles to compute deflection and have proper slopes.
- Use a physics engine (e.g., Box2D)

Obviously, the second is more popular (though I suspect that Braid went for the first), both because it is easier and because it allows you to do many other things with physics in the game. Unfortunately, in my opinion, one has to be very careful when going this route, to avoid making the game feel like a generic, uninteresting physics-platformer.

Compound objects

This approach has its own unique problems. It may suddenly be difficult to tell whether the player is actually standing on the floor (due to rounding errors), or whether it's hitting a wall or sliding down a steep incline. If using a physics engine, friction can be an issue, as you'll want friction to be high on the foot but low on the sides.

There are different ways to deal with those, but a popular solution is to divide the character into several different polygons, each with different roles associated: so you'd (optionally) have the main central body, then a thin rectangle for feet, and two thin rectangles for sides, and another for head or some similar combination. Sometimes they are tapered to avoid getting caught in obstacles. They can have different physics properties, and collision callbacks on those can be used to determine the status of the character. For more information, sensors (non-colliding objects that are just used to check for overlap) can be used. Common cases include determining whether we're close enough to the floor to perform a jump, or if the character is pushing against a wall, etc.

GENERAL CONSIDERATIONS

Regardless of the type of platform movement that you have chosen (except perhaps for type #1), a few general considerations apply.

Acceleration

Super Mario World (low acceleration), Super Metroid (mid acceleration), and Mega Man 7 (high acceleration)

One of the factors that affects the feel of a platform the most is the acceleration of the character. Acceleration is the rate of change in speed. When it is low, the character takes a long time to reach its maximum velocity or to come to a halt after the player lets go of controls. This makes the character feel "slippery," and can be hard to master. This movement is most commonly associated with the Super Mario series of games. When the acceleration is high, the character takes very little (or no time) to go from zero to maximum speed and back, resulting in very fast responding "twitchy" controls, as seen in the Mega Man series (I believe that Mega Man actually employs infinite acceleration, that is, you're either stopped or on full speed).

Even if a game has no acceleration on its horizontal movement, it is likely to have at least some for the jump arcs — otherwise they will be shaped like triangles.

How it works

Implementing acceleration is actually fairly simple, but there are a few traps to watch out for.

- Determine xTargetSpeed. This should be 0 if the player is not touching the controls, -maxSpeed if pressing left or +maxSpeed if pressing right.
- Determine yTargetSpeed. This should be 0 if the player is standing on a platform, +terminalSpeed otherwise.
- For each axis, accelerate the current speed toward target speed using either weighted averaging or adding acceleration.

The two acceleration methods are as follows:

- Weighted averaging: acceleration is a number ("a") from 0 (no change) to 1 (instant acceleration). Use that value to linearly interpolate between target and current speed, and set the result as current speed.

```
vector2f curSpeed = a * targetSpeed + (1-a) * curSpeed;
if (fabs(curSpeed.x) < threshold) curSpeed.x = 0;
if (fabs(curSpeed.y) < threshold) curSpeed.y = 0;
```

- Adding acceleration: We'll determine which direction to add the acceleration to (using the sign function, which returns 1 for numbers >0 and -1 for <0), then check if we overshot.

```
vector2f direction = vector2f(sign(targetSpeed.x - curSpeed.x),
                              sign(targetSpeed.y - curSpeed.y));
curSpeed += acceleration * direction;
if (sign(targetSpeed.x - curSpeed.x) != direction.x)
    curSpeed.x = targetSpeed.x;
if (sign(targetSpeed.y - curSpeed.y) != direction.y)
    curSpeed.y = targetSpeed.y;
```

It's important to integrate the acceleration into the speed before moving the character; otherwise, you'll introduce a one-frame lag into character input.

When the character hits an obstacle, it's a good idea to zero his speed along that axis.

Jump control

Super Metroid - Samus performing the "Space Jump" (with "Screw Attack" power-up)

Jumping in a platform game can be as simple as checking if the player is on the ground (or, often, whether he was on the ground anytime in the last n frames), and, if so, giving the character an initial negative y speed (in physical terms, an impulse) and letting gravity do the rest.

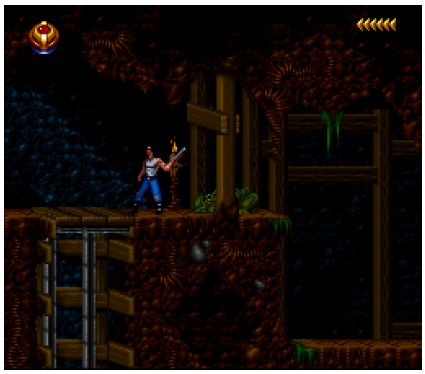
There are four general ways in which the player can control the jump:

- Impulse: seen in games such as Super Mario World and Sonic the Hedgehog, the jump preserves the momentum (that is, in implementation terms, the speed) that the character had before the jump. In some games, this is the only way to influence the arc of the jump — just like in real life. There is nothing to implement here — it will be like this unless you do something to stop it!
- Aerial acceleration: that is, retaining control of horizontal movement while in midair. Though this is physically implausible, it is a very popular feature, as it makes the character much more controllable. Almost every platformer game has it, with exceptions for games similar to Prince of Persia. Generally, the airborne acceleration is greatly reduced, so impulse is important, but some games (like Mega Man) give you full air control. This is generally implemented as merely tweaking the acceleration parameter while you're airborne.
- Ascent control: another physically implausible action, but very popular, as it gives you much greater control over the character. The longer you hold the jump button, the higher the character jumps. Typically, this

is implemented by continuing to add impulse to the character (though this impulse can incrementally decrease) for as long as the button is held, or alternatively by suppressing gravity while the button is held. A time limit is imposed, unless you want the character to be able to jump infinitely.

- Multiple jumps: once airborne, some games allow the player to jump again, perhaps for an unlimited number of times (as in the Space Jump in Super Metroid or the flight in Talbot's Odyssey), or for a limited number of jumps before touching the ground ("double jump" being the most common choice). This can be accomplished by keeping a counter that increases for each jump and decreases when you're on the ground (be careful when you update this, or you might reset it right after the first jump), and only allowing further jumps if the counter is low enough. Sometimes, the second jump is shorter than the initial one. Other restrictions may apply — the Space Jump only triggers if you're already doing a spin jump and just began to fall.

Animations and leading



Black Thorne, character doing a long animation before shooting backward (Y button)

In many games, your character will play an animation before actually performing the action you requested. However, on a twitchy action-based game, this will frustrate players — DON'T DO THAT! You should still have leading animations for things such as jumping and running, but if you care about how the game responds, make those cosmetic only, with the action taken immediately regardless of the animation.

Smoother movement

Using integers to represent the position of the characters is wise, as it makes movement faster and more stable. However, if you use integers for everything, you will end up with some jerky motion. There are multiple solutions to this. These are a few:

- Use a float for all computations and for storing position, and cast to integer whenever you're rendering or computing collisions. Fast and simple, but it starts losing precision if you move too far away from (0,0). This is probably not relevant unless you have a very large playfield, but it's something to keep in mind. If it comes to it, you can use a double instead.
- Use a fixed point number for all computations and position, and again cast to integer when you're rendering or computing collisions. Less precise than float and with a more limited range, but the precision is uniform and can, on some hardware, be faster (notably, floating point processing is slow on many mobile phones).
- Store position as an integer, but keep a "remainder" stored in a float. When integrating position, compute the delta-movement as a float; add the remainder to the delta-movement, and then add the integer part of this value to the position and the fractional part of the "remainder" field. On the next frame, the remainder will get added back in. The advantage of this method is that you're using an integer everywhere except for movement, ensuring that you won't have floating-point complications elsewhere, and increasing performance. This technique is also very suitable if you have some framework in which the position of the object has to be an integer, or where it is a float, but that same position is used directly by the rendering system — in that case, you can use the framework-provided float position to store integer values only, to make sure that the rendering is always aligned to pixels. ■

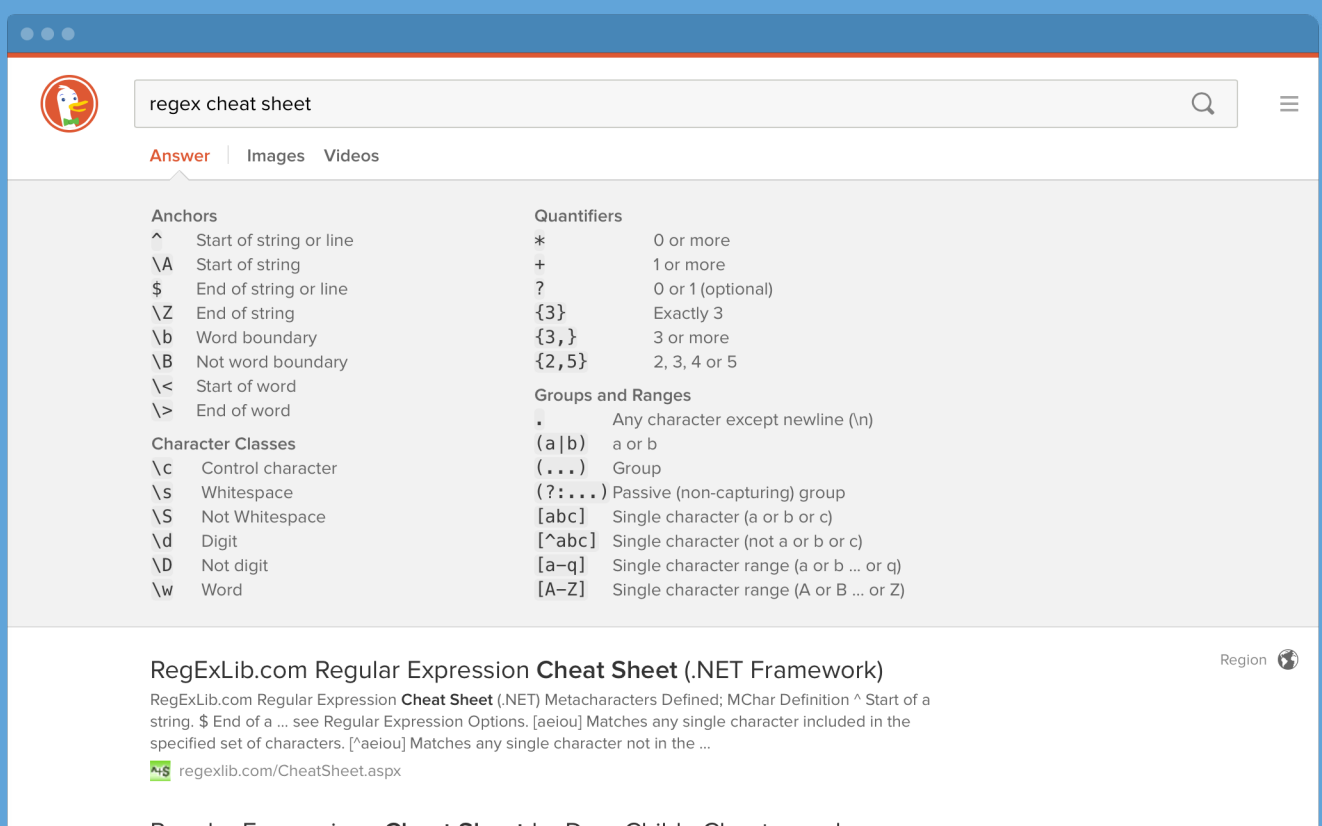
Rodrigo is a Senior Games Developer at Bossa Studios, in London. Originally from Brazil, he started making games at the age of 11, and later came to Europe to work in the video game industry.

Reprinted with permission of the original author. First appeared in *hn.my/2d* (higherorderfun.com)

Join the DuckDuckGo Open Source Community.

Create Instant Answers or share ideas and help change the future of search.

Featured IA: Regex Contributor: mintsoft
Get started at duckduckhack.com



How a Kalman Filter Works, in Pictures

By TIM BABB

I HAVE TO TELL you about the Kalman filter, because what it does is pretty damn amazing.

Surprisingly few software engineers and scientists seem to know about it, and that makes me sad because it is such a general and powerful tool for combining information in the presence of uncertainty. At times its ability to extract accurate information seems almost magical — and if it sounds like I'm talking this up too much, then take a look at this video [hn.my/imu] where I demonstrate a Kalman filter figuring out the orientation of a free-floating body by looking at its velocity. Totally neat!

What is it?

You can use a Kalman filter in any place where you have uncertain information about some dynamic system, and you can make an educated guess about what the system is going to do next. Even if messy reality comes along and interferes with the clean motion you guessed about, the Kalman filter will often do a very good job of figuring out what actually happened. And it can take advantage of correlations between crazy phenomena that you maybe wouldn't have thought to exploit!

Kalman filters are ideal for systems which are continuously changing. They have the advantage that they are light on memory (they don't need to keep any history other than the previous state), and they are very fast, making them well suited for real time problems and embedded systems.

The math for implementing the Kalman filter appears pretty scary and opaque in most places you find on Google. That's a bad state of affairs, because the Kalman filter is actually super simple and easy to understand if you look at it in the right way. Thus it makes a great article topic, and I will attempt to illuminate it with lots of clear, pretty pictures and colors. The prerequisites are simple: all you need is a basic understanding of probability and matrices.

I'll start with a loose example of the kind of thing a Kalman filter can solve, but if you want to get right to the shiny pictures and math, feel free to jump ahead.

What can we do with a Kalman filter?

Let's make a toy example. You've built a little robot that can wander around in the woods, and the robot needs to know exactly where it is so that it can navigate.

We'll say our robot has a state $\vec{x_k}$, which is just a position and a velocity:

$$\vec{x_k} = (\vec{p}, \vec{v})$$

Note that the state is just a list of numbers about the underlying configuration of your system; it could be anything. In our example it's position and velocity, but it could be data about the amount of fluid in a tank, the temperature of a car engine, the position of a user's finger on a touchpad, or any number of things you need to keep track of.

Our robot also has a GPS sensor, which is accurate to about 10 meters, which is good, but it needs to know its location more precisely than 10 meters. There are lots of gullies and cliffs in these woods, and if the robot is wrong by more than a few feet, it could fall off a cliff. So GPS by itself is not good enough.

We might also know something about how the robot moves: It knows the commands sent to the wheel motors, and it knows that if it's headed in one direction and nothing interferes, at the next instant it will likely be further along that same direction. But of course it doesn't know everything about its motion: it might be buffeted by the wind, the wheels might slip a little bit, or roll over bumpy terrain. So the amount the wheels have turned might not exactly represent how far the robot has actually traveled, and the prediction won't be perfect.

The GPS sensor tells us something about the state, but only indirectly, and with some uncertainty or inaccuracy. Our prediction tells us something about how the robot is moving, but only indirectly, and with some uncertainty or inaccuracy.

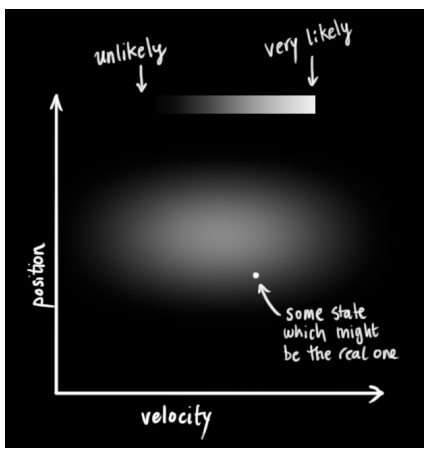
But if we use all the information available to us, can we get a better answer than either estimate would give us by itself? Of course the answer is yes, and that's what a Kalman filter is for.

How a Kalman filter sees your problem

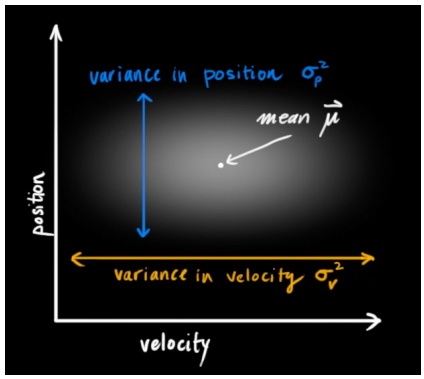
Let's look at the landscape we're trying to interpret. We'll continue with a simple state having only position and velocity.

$$\vec{x} = \begin{bmatrix} p \\ v \end{bmatrix}$$

We don't know what the actual position and velocity are; there are a whole range of possible combinations of position and velocity that might be true, but some of them are more likely than others:

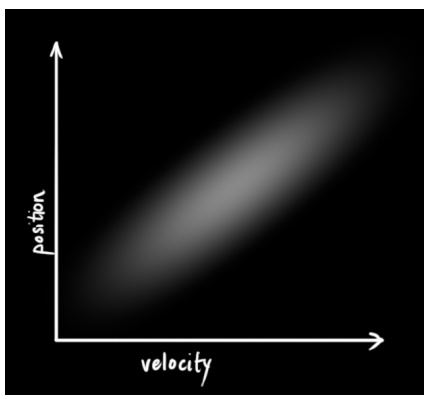


The Kalman filter assumes that both variables (position and velocity, in our case) are random and Gaussian distributed. Each variable has a mean value μ, which is the center of the random distribution (and its most likely state), and a variance σ^2, which is the uncertainty:



In the above picture, position and velocity are uncorrelated, which means that the state of one variable tells you nothing about what the other might be.

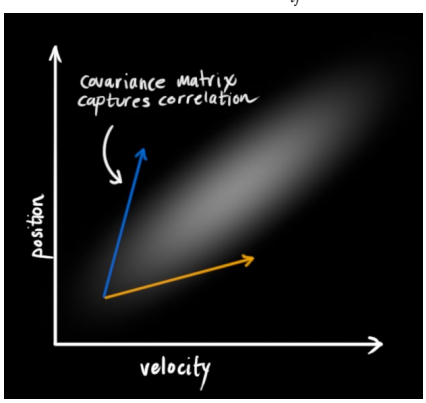
The example below shows something more interesting: Position and velocity are correlated. The likelihood of observing a particular position depends on what velocity you have:



This kind of situation might arise if, for example, we are estimating a new position based on an old one. If our velocity was high, we probably moved farther, so our position will be more distant. If we're moving slowly, we didn't get as far.

This kind of relationship is really important to keep track of, because it gives us more information — one measurement tells us something about what the others could be. And that's the goal of the Kalman filter: we want to squeeze as much information from our uncertain measurements as we possibly can!

This correlation is captured by something called a covariance matrix. In short, each element of the matrix Σ_{ij} is the degree of correlation between the ith state variable and the jth state variable. (You might be able to guess that the covariance matrix is symmetric, which means that it doesn't matter if you swap i and j). Covariance matrices are often labelled "Σ", so we call their elements "Σ_{ij}".



Describing the problem with matrices

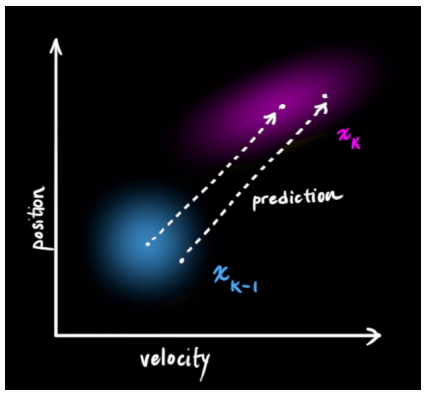
We're modeling our knowledge about the state as a Gaussian blob, so we need two pieces of information at time k: We'll call our best estimate $\hat{\mathbf{x}}_k$ (the mean, elsewhere named μ), and its covariance matrix P_k.

$$\hat{\mathbf{x}}_k = \begin{bmatrix} \text{position} \\ \text{velocity} \end{bmatrix}$$

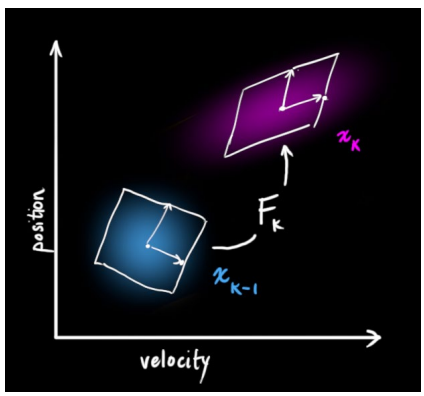
$$\mathbf{P}_k = \begin{bmatrix} \Sigma_{pp} & \Sigma_{pv} \\ \Sigma_{vp} & \Sigma_{vv} \end{bmatrix} \quad (1)$$

(Of course we are using only position and velocity here, but it's useful to remember that the state can contain any number of variables and represent anything you want).

Next, we need some way to look at the current state (at time k-1) and predict the next state at time k. Remember, we don't know which state is the "real" one, but our prediction function doesn't care. It just works on all of them and gives us a new distribution:



We can represent this prediction step with a matrix, F_k:



It takes every point in our original estimate and moves it to a new predicted location, which is where the system would move if that original estimate was the right one.

Let's apply this. How would we use a matrix to predict the position and velocity at the next moment in the future? We'll use a really basic kinematic formula:

$$\begin{aligned} p_k &= p_{k-1} + \Delta t \quad v_{k-1} \\ v_k &= \quad v_{k-1} \end{aligned}$$

In other words:

$$\hat{\mathbf{x}}_k = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \hat{\mathbf{x}}_{k-1} \quad (2)$$

$$= \mathbf{F}_k \hat{\mathbf{x}}_{k-1} \quad (3)$$

We now have a prediction matrix which gives us our next state, but we still don't know how to update the covariance matrix.

This is where we need another formula. If we multiply every point in a distribution by a matrix A, then what happens to its covariance matrix Σ?

Well, it's easy. I'll just give you the identity:

$$\begin{aligned} Cov(x) &= \Sigma \\ Cov(\mathbf{A}x) &= \mathbf{A}\Sigma\mathbf{A}^T \end{aligned} \quad (4)$$

So combining (4) with equation (3):

$$\begin{aligned} \hat{\mathbf{x}}_k &= \mathbf{F}_k \hat{\mathbf{x}}_{k-1} \\ \mathbf{P}_k &= \mathbf{F}_k \mathbf{P}_{k-1} \mathbf{F}_k^T \end{aligned} \quad (5)$$

External influence

We haven't captured everything, though. There might be some changes that aren't related to the state itself — the outside world could be affecting the system.

For example, if the state models the motion of a train, the train operator might push on the throttle, causing the train to accelerate. Similarly, in our robot example, the navigation software might issue a command to turn the wheels or stop. If we know this additional information about what's going on in the world, we could stuff it into a vector called $\vec{\mathbf{u}_k}$, do something with it, and add it to our prediction as a correction.

Let's say we know the expected acceleration a due to the throttle setting or control commands. From basic kinematics we get:

$$p_k = p_{k-1} + \Delta t \quad v_{k-1} + \frac{1}{2} a \Delta t^2$$

$$v_k = \quad v_{k-1} + a\Delta t$$

In matrix form:

$$\hat{\mathbf{x}}_k = \mathbf{F}_k \hat{\mathbf{x}}_{k-1} + \begin{bmatrix} \frac{\Delta t^2}{2} \\ \Delta t \end{bmatrix} a \qquad (6)$$

$$= \mathbf{F}_k \hat{\mathbf{x}}_{k-1} + \mathbf{B}_k \vec{\mathbf{u}_k}$$

B_k is called the control matrix and $\vec{\mathbf{u}_k}$ the control vector. (For very simple systems with no external influence, you could omit these).

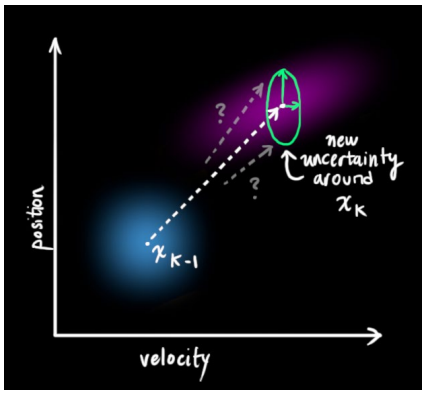
Let's add one more detail. What happens if our prediction is not a 100% accurate model of what's actually going on?

External uncertainty

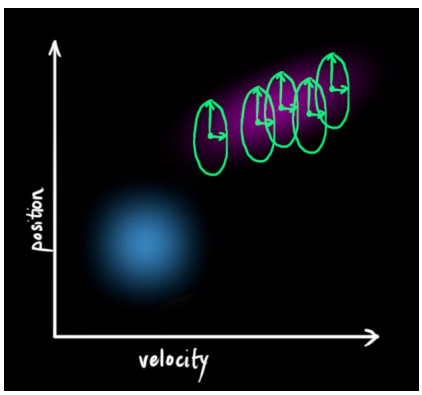
Everything is fine if the state evolves based on its own properties. Everything is still fine if the state evolves based on external forces, so long as we know what those external forces are.

But what about forces that we don't know about? If we're tracking a quadcopter, for example, it could be buffeted around by wind. If we're tracking a wheeled robot, the wheels could slip, or bumps on the ground could slow it down. We can't keep track of these things, and if any of this happens, our prediction could be off because we didn't account for those extra forces.

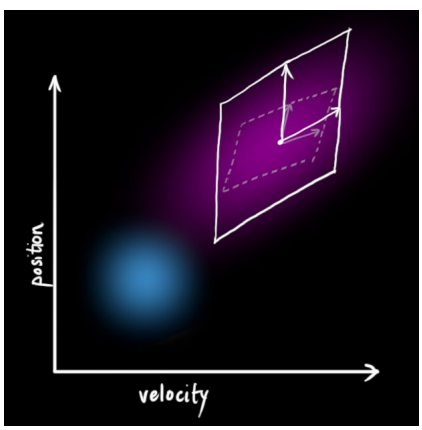
We can model the uncertainty associated with the "world" (i.e., things we aren't keeping track of) by adding some new uncertainty after every prediction step:



Every state in our original estimate could have moved to a range of states. Because we like Gaussian blobs so much, we'll say that each point in $\hat{\mathbf{x}}_{k-1}$ is moved to somewhere inside a Gaussian blob with covariance Q_k. Another way to say this is that we are treating the untracked influences as noise with covariance Q_k.



This produces a new Gaussian blob, with a different covariance (but the same mean):



We get the expanded covariance by simply adding Q_k, giving our complete expression for the prediction step:

$$\hat{\mathbf{x}}_k = \mathbf{F}_k \hat{\mathbf{x}}_{k-1} + \mathbf{B}_k \vec{\mathbf{u}_k} \qquad (7)$$

$$\mathbf{P}_k = \mathbf{F_k} \mathbf{P}_{k-1} \mathbf{F}_k^T + \mathbf{Q}_k$$

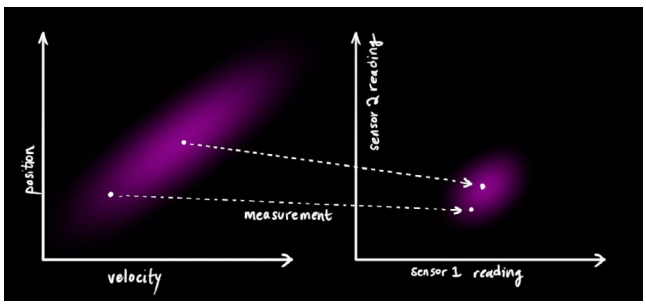
In other words, the new best estimate is a prediction made from previous best estimate, plus a correction for known external influences.

And the new uncertainty is predicted from the old uncertainty, with some additional uncertainty from the environment.

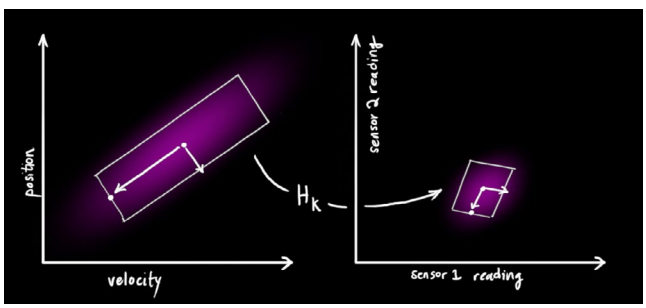
All right, so that's easy enough. We have a fuzzy estimate of where our system might be, given by $\hat{\mathbf{x}}_\mathbf{k}$ and P_k. What happens when we get some data from our sensors?

Refining the estimate with measurements

We might have several sensors which give us information about the state of our system. For the time being it doesn't matter what they measure. Perhaps one reads position and the other reads velocity. Each sensor tells us something indirect about the state — in other words, the sensors operate on a state and produce a set of readings.



Notice that the units and scale of the reading might not be the same as the units and scale of the state we're keeping track of. You might be able to guess where this is going: We'll model the sensors with a matrix, H_k.

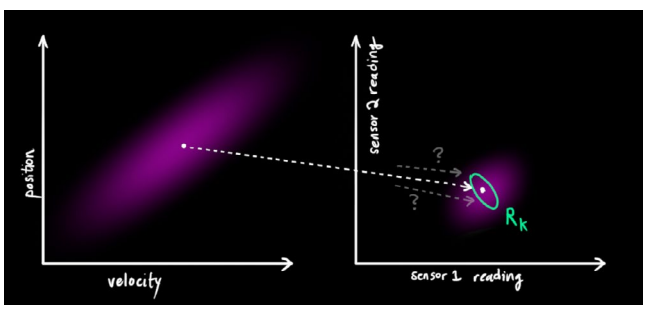


We can figure out the distribution of sensor readings we'd expect to see in the usual way:

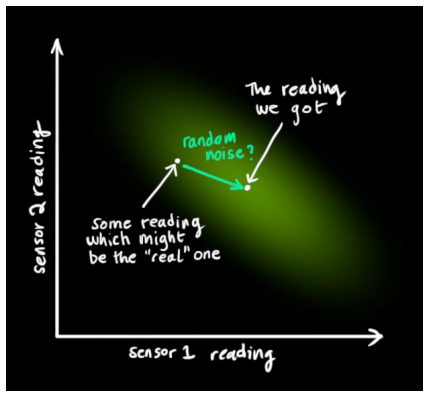
$$\vec{\mu}_{\text{expected}} = \mathbf{H}_k\hat{\mathbf{x}}_k \tag{8}$$

$$\mathbf{\Sigma}_{\text{expected}} = \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T$$

One thing that Kalman filters are great for is dealing with sensor noise. In other words, our sensors are at least somewhat unreliable, and every state in our original estimate might result in a range of sensor readings.

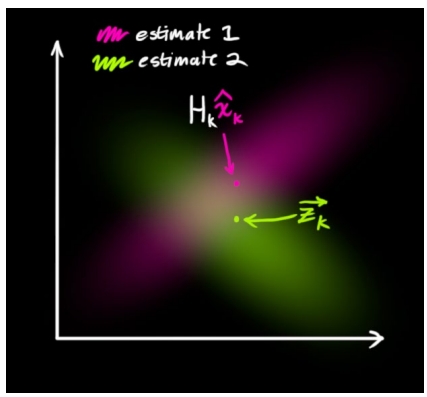


From each reading we observe, we might guess that our system was in a particular state. But because there is uncertainty, some states are more likely than others to have produced the reading we saw:



We'll call the covariance of this uncertainty (i.e., of the sensor noise) R_k. The distribution has a mean equal to the reading we observed, which we'll call $\vec{\mathbf{z}}_k$.

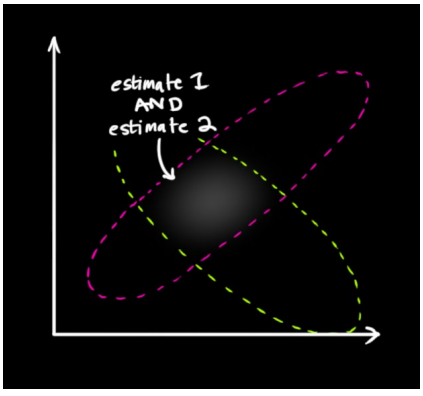
So now we have two Gaussian blobs: One surrounding the mean of our transformed prediction, and one surrounding the actual sensor reading we got.



We must try to reconcile our guess about the readings we'd see based on the predicted state(pink) with a different guess based on our sensor readings (green) that we actually observed.

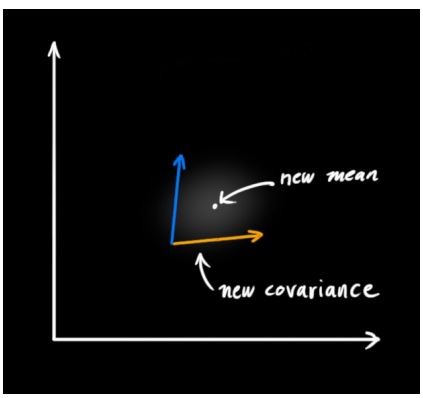
So what's our new most likely state? For any possible reading (z1, z2), we have two associated probabilities: (1) The probability that our sensor reading $\vec{\mathbf{z}}_k$ is a (mis-)measurement of (z1, z2), and (2) the probability that our previous estimate thinks (z1, z2) is the reading we should see.

If we have two probabilities and we want to know the chance that both are true, we just multiply them together. So, we take the two Gaussian blobs and multiply them:



What we're left with is the overlap, the region where both blobs are bright/likely. And it's a lot more precise than either of our previous estimates. The mean of this distribution is the configuration for which both estimates are most likely, and is therefore the best guess of the true configuration given all the information we have.

Hmm. This looks like another Gaussian blob.



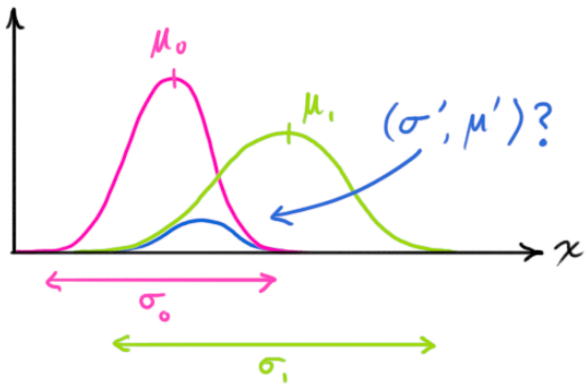
As it turns out, when you multiply two Gaussian blobs with separate means and covariance matrices, you get a new Gaussian blob with its own mean and covariance matrix! Maybe you can see where this is going: there's got to be a formula to get those new parameters from the old ones!

Combining Gaussians

Let's find that formula. It's easiest to look at this first in one dimension. A 1D Gaussian bell curve with variance σ^2 and mean μ is defined as:

$$\mathcal{N}(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{9}$$

We want to know what happens when you multiply two Gaussian curves together:



$$\mathcal{N}(x, \mu_0, \sigma_0) \cdot \mathcal{N}(x, \mu_1, \sigma_1) \stackrel{?}{=} \mathcal{N}(x, \mu, \sigma) \tag{10}$$

You can substitute equation (9) into equation (10) and do some algebra (being careful to renormalize, so that the total probability is 1) to obtain:

$$\mu = \mu_0 + \frac{\sigma_0^2(\mu_1 - \mu_0)}{\sigma_0^2 + \sigma_1^2} \tag{11}$$

$$\sigma^2 = \sigma_0^2 \frac{\sigma_0^4}{\sigma_0^2 + \sigma_1^2}$$

We can simplify by factoring out a little piece and calling it k:

$$\mathbf{k} = \frac{\sigma_0^2}{\sigma_0^2 + \sigma_1^2} \tag{12}$$

$$\begin{aligned} \mu &= \mu_0 + \mathbf{k}(\mu_1 - \mu_0) \\ \sigma^2 &= \sigma_0^2 - \mathbf{k}\sigma_0^2 \end{aligned} \tag{13}$$

Take note of how you can take your previous estimate and add something to make a new estimate. And look at how simple that formula is!

But what about a matrix version? Well, let's just re-write equations (12) and (13) in matrix form. If Σ is the covariance matrix of a Gaussian blob, and $\vec{\mu}$ is its mean along each axis, then:

$$\mathbf{K} = \Sigma_0(\Sigma_0 + \Sigma_1)^{-1} \tag{14}$$

$$\begin{aligned} \vec{\mu} &= \vec{\mu_0} + \mathbf{K}(\vec{\mu_1} - \vec{\mu_0}) \\ \Sigma &= \Sigma_0 - \mathbf{K}\Sigma_0 \end{aligned} \tag{15}$$

K is a matrix called the Kalman gain, and we'll use it in just a moment.

Easy! We're almost finished!

Putting it all together

We have two distributions: the predicted measurement with , $(\mu_0, \Sigma_0) = (\mathbf{H}_k\hat{\mathbf{x}}_k, \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T)$ and the observed measurement with $(\mu_1, \Sigma_1) = (\vec{\mathbf{z}_k}, \mathbf{R}_k)$. We can just plug these into equation (15) to find their overlap:

$$\begin{aligned} \mathbf{H}_k\hat{\mathbf{x}}_k &= \mathbf{H}_k\hat{\mathbf{x}}_k &&+\mathbf{K}(\vec{\mathbf{z}_k} - \mathbf{H}_k\hat{\mathbf{x}}_k) \\ \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T &= \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T &&-\mathbf{K}\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T \end{aligned} \quad (16)$$

And from (14), the Kalman gain is:

$$\mathbf{K} = \mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T(\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T + \mathbf{R}_k)^{-1} \quad (17)$$

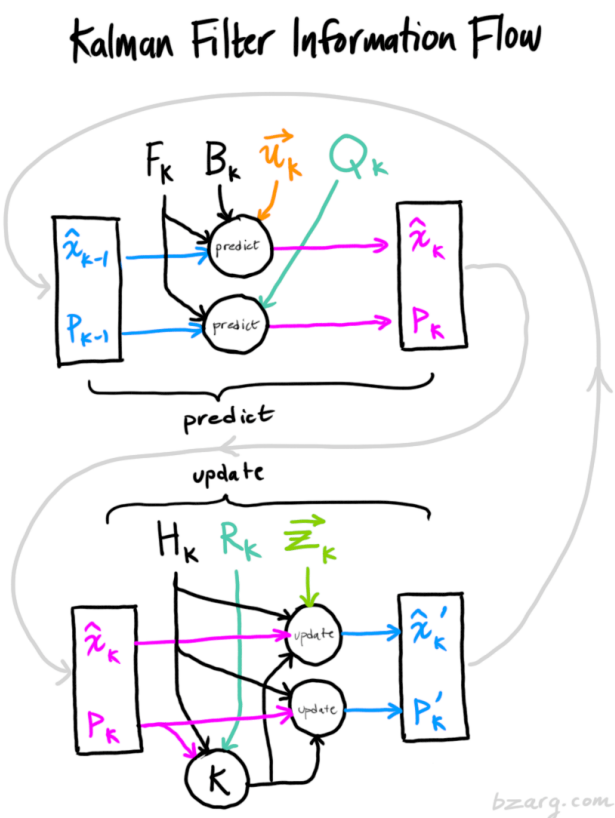
We can knock an H_k off the front of every term in (16) and (17) (note that one is hiding inside K), and an H_k^T off the end of all terms in the equation for $\mathrm{P'}_k$.

$$\begin{aligned} \hat{\mathbf{x}}_k &= \hat{\mathbf{x}}_k &&+\mathbf{K}(\vec{\mathbf{z}_k} - \mathbf{H}_k\hat{\mathbf{x}}_k) \\ \mathbf{P}_k &= \mathbf{P}_k &&-\mathbf{K}\mathbf{H}_k\mathbf{P}_k \end{aligned} \quad (18)$$

$$\mathbf{K} = \mathbf{P}_k\mathbf{H}_k^T(\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T + \mathbf{R}_k)^{-1} \quad (19)$$

…giving us the complete equations for the update step.

And that's it! $\hat{\mathbf{x}}'_k$ is our new best estimate, and we can go on and feed it (along with $\mathrm{P'}_k$) back into another round of predict or update as many times as we like.



Wrapping up

Of all the math above, all you need to implement are equations (7), (18), and (19). (Or if you forget those, you could re-derive everything from equations (4) and (15).)

This will allow you to model any linear system accurately. For nonlinear systems, we use the extended Kalman filter, which works by simply linearizing the predictions and measurements about their mean.

If I've done my job well, hopefully someone else out there will realize how cool these things are and come up with an unexpected new place to put them into action. ■

Some credit and referral should be given to this fine document, [hn.my/kalman2] which uses a similar approach involving overlapping Gaussians. More in-depth derivations can be found there, for the curious.

Tim Babb is a software engineer at Pixar Animation Studios, where he works on feature films fixing things and building light transport code. In his spare time, he builds projects relating to computational geometry, atmospheric optics, poker, and sensor fusion, and maintains a blog about math and technology.

Reprinted with permission of the original author.
First appeared in *hn.my/kalman* (bzarg.com)

Introduction to Monte Carlo Tree Search

By JEFF BRADBERRY

The subject of the game AI generally begins with is so-called perfect information games. These are turn-based games where the players have no information hidden from each other, and there is no element of chance in the game mechanics (such as by rolling dice or drawing cards from a shuffled deck). Tic-Tac-Toe, Connect 4, checkers, Reversi, chess, and Go are all games of this type. Because everything in this type of game is fully determined, a tree can, in theory, be constructed that contains all possible outcomes, and a value assigned corresponding to a win or a loss for one of the players. Finding the best possible play, then, is a matter of doing a search on the tree, with the method of choice at each level alternating between picking the maximum value and picking the minimum value, matching the different players' conflicting goals as the search proceeds down the tree. This algorithm is called Minimax. [hn.my/minimax]

The problem with Minimax, though, is that it can take an impractical amount of time to do a full search of the game tree. This is particularly true for games with a high branching factor, or high average number of available moves per turn. This is because the basic version of Minimax needs to search all of the nodes in the tree to find the optimal solution, and the number of nodes in the tree that must be checked grows exponentially with the branching factor. There are methods of mitigating this problem, such as searching only to a limited number of moves ahead (or ply) and then using an evaluation function to estimate the value of the position, or by pruning branches to be searched if they are unlikely to be worthwhile. Many of these techniques, though, require encoding domain knowledge about the game, which may be difficult to gather or formulate. And while such methods have produced chess programs capable of defeating grand masters, similar success in Go has been elusive, particularly for programs playing on the full 19x19 board.

However, there is a game AI technique that does do well for games with a high branching factor, and it has come to dominate the field of Go playing programs. It is easy to create a basic implementation of this algorithm that will give good results for games with a smaller branching factor, and relatively simple adaptations can build on it and improve it for games like chess or Go. It can be configured to stop after any desired amount of time, with longer times resulting in stronger game play. Since it doesn't necessarily require game-specific knowledge, it can be used for general game playing. It may even be adaptable to games that incorporate randomness in the rules. This technique is called Monte Carlo Tree Search. In this article I will describe how MCTS works, specifically a variant called Upper Confidence bound applied to Trees (UCT), and then will show you how to build a basic implementation in Python.

Imagine, if you will, that you are faced with a row of slot machines, each with different payout probabilities and amounts. As a rational person (if you are going to play

them at all), you would prefer to use a strategy that will allow you to maximize your net gain. But how can you do that? For whatever reason, there is no one nearby, so you can't watch someone else play for a while to gain information about which is the best machine. Clearly, your strategy is going to have to balance playing all of the machines to gather that information yourself, with concentrating your plays on the observed best machine. One strategy, called UCB1, does this by constructing statistical confidence intervals for each machine.

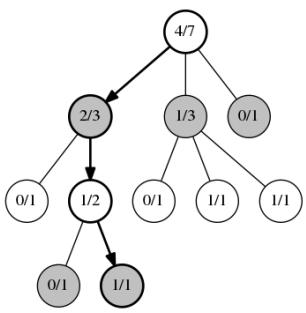
$$\bar{x}_i \pm \sqrt{\frac{2 \ln n}{n_i}}$$

where:

- $\bar{x}_i$: the mean payout for machine i
- n_i: the number of plays of machine i
- n: the total number of plays

Then, your strategy is to pick the machine with the highest upper bound each time. As you do so, the observed mean value for that machine will shift, and its confidence interval will become narrower, but all of the other machines' intervals will widen. Eventually, one of the other machines will have an upper bound that exceeds that of your current one, and you will switch to that one. This strategy has the property that you regret the difference between what you would have won by playing solely on the actual best slot machine and your expected winnings under the strategy that you do use -- grows only as $\mathcal{O}(\ln n)$. This is the same big-O growth rate as the theoretical best for this problem (referred to as the multi-armed bandit problem), and has the additional benefit of being easy to calculate.

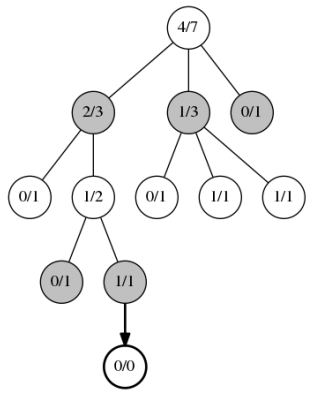
And here's how Monte Carlo comes in. In a standard Monte Carlo process, a large number of random simulations are run, in this case, from the board position that you want to find the best move for. Statistics are kept for each possible move from this starting state, and then the move with the best overall results is returned. The downside to this method, though, is that for any given turn in the simulation, there may be many possible moves, but only one or two that are good. If a random move is chosen each turn, it becomes extremely unlikely that the simulation will hit upon the best path forward. So, UCT has been proposed as an enhancement. The idea is this: Any given board position can be considered a multi-armed bandit problem, if statistics are available for all of the positions that are only one move away. So instead of doing many purely random simulations, UCT works by doing many multi-phase playouts.



Selection

Here the positions and moves selected by the UCB1 algorithm at each step are marked in bold. Note that a number of playouts have already been run to accumulate the statistics shown. Each circle contains the number of wins / number of times played.

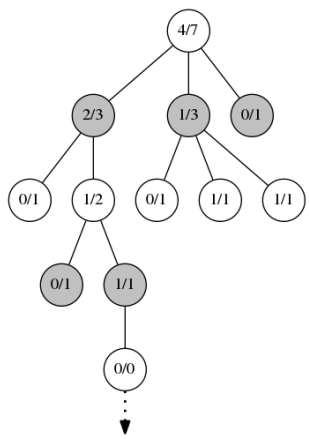
The first phase, selection, lasts while you have the statistics necessary to treat each position you reach as a multi-armed bandit problem. The move to use, then, would be chosen by the UCB1 algorithm instead of randomly, and applied to obtain the next position to be considered. Selection would then proceed until you reach a position where not all of the child positions have statistics recorded.



Expansion

The position marked 1/1 at the bottom of the tree has no further statistics records under it, so we choose a random move and add a new record for it (bold), initialized to 0/0.

The second phase, expansion, occurs when you can no longer apply UCB1. An unvisited child position is randomly chosen, and a new record node is added to the tree of statistics.

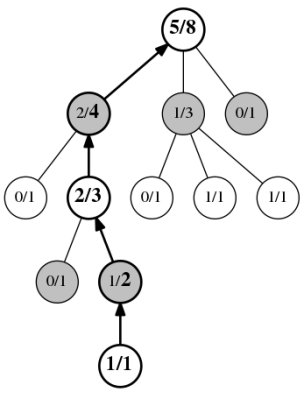


Simulation

Once the new record is added, the Monte Carlo simulation begins, here depicted with a dashed arrow. Moves in the simulation may be completely

random, or may use calculations to weight the randomness in favor of moves that may be better.

After expansion occurs, the remainder of the playout is in phase 3, simulation. This is done as a typical Monte Carlo simulation, either purely random or with some simple weighting heuristics if a light playout is desired, or by using some computationally expensive heuristics and evaluations for a heavy playout. For games with a lower branching factor, a light playout can give good results.



Back-Propagation
After the simulation reaches an end, all of the records in the path taken are updated. Each has his play count incremented by one, and each that matches the winner has its win count incremented by one, here shown by the bolded numbers.

Finally, the fourth phase is the update or back-propagation phase. This occurs when the playout reaches the end of the game. All of the positions visited during this playout have their play count incremented, and if the player for that position won the playout, the win count is also incremented.

This algorithm may be configured to stop after any desired length of time, or on some other condition. As more and more playouts are run, the tree of statistics grows in memory and the move that will finally be chosen will converge towards the actual optimal play, though that may take a very long time, depending on the game.

For more details about the mathematics of UCB1 and UCT, see Finite-time Analysis of the Multi-armed Bandit Problem and Bandit-based Monte Carlo Planning.

Now let's see some code. To separate concerns, we're going to need a Board class, whose purpose is to encapsulate the rules of a game and which will care nothing about the AI, and a Monte Carlo class, which will only care about the AI algorithm and will query into the Board object in order to obtain information about the game. Let's assume a Board class supporting this interface:

```
class Board(object):
    def start(self):
        # Returns a representation of the
        # starting state of the game.
        pass

    def current_player(self, state):
        # Takes the game state and returns the
        # current player's number.
        pass

    def next_state(self, state, play):
        # Takes the game state, and the move to
        # be applied.
        # Returns the new game state.
        pass

    def legal_plays(self, state_history):
        # Takes a sequence of game states
        # representing the full game history,
        # and returns the full
        # list of moves that are legal plays for
        # the current player.
        pass

    def winner(self, state_history):
        # Takes a sequence of game states
        # representing the full game history.
        # If the game is now won, return the
        # player number. If the game is still
        # ongoing, return zero.  If the game is
        # tied, return a different distinct
        # value, e.g. -1.
        pass
```

For the purposes of this article, I'm not going to flesh this part out any further, but for example code, you can find one of my implementations on github. However, it is important to note that we will require that the state data structure is hashable and equivalent states hash to the same value. I personally use flat tuples as my state data structures.

The AI class we will be constructing will support this interface:

```
class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # Takes an instance of a Board and
        # optionally some keyword arguments.
        # Initializes the list of game states @
        # and the statistics tables.
        pass

    def update(self, state):
        # Takes a game state, and appends it to
        # the history.
        pass

    def get_play(self):
        # Causes the AI to calculate the best
        # move from the
        # current game state and return it.
        pass

    def run_simulation(self):
        # Plays out a "random" game from the
        # current position, then updates the
        # statistics tables with the result.
        pass
```

Let's begin with the initialization and bookkeeping. The board object is what the AI will be using to obtain information about where the game is going and what the AI is allowed to do, so we need to store it. Additionally, we need to keep track of the state data as we get it.

```
class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        self.board = board
        self.states = []

    def update(self, state):
        self.states.append(state)
```

The UCT algorithm relies on playing out multiple games from the current state, so let's add that next.

```
Import datetime

class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        seconds = kwargs.get('time', 30)
        self.calculation_time = datetime.
timedelta(seconds=seconds)

    # ...

    def get_play(self):
        begin = datetime.datetime.utcnow()
        while datetime.datetime.utcnow() - begin
< self.calculation_time:
            self.run_simulation()
```

Here we've defined a configuration option for the amount of time to spend on a calculation, and get_play will repeatedly call run_simulation until that amount of time has passed. This code won't do anything particularly useful yet because we still haven't defined run_simulation, so let's do that now.

```
# ...
from random import choice

class MonteCarlo(object):
  def __init__(self, board, **kwargs):
    # ...
    self.max_moves = kwargs.get('max_moves',
100)

  # ...

  def run_simulation(self):
    states_copy = self.states[:]
    state = states_copy[-1]

      for t in xrange(self.max_moves):
        legal = self.board.legal_plays(states_
copy)

        play = choice(legal)
        state = self.board.next_state(state, play)
        states_copy.append(state)

        winner = self.board.winner(states_copy)
        if winner:
          break
```

This adds the beginnings of the run_simulation method, which either selects a move using UCB1 or chooses a random move from the set of legal moves each turn until the end of the game. We have also introduced a configuration option for limiting the number of moves forward that the AI will play.

You may notice at this point that we are making a copy of self.states and adding new states to it, instead of adding directly to self.states. This is because self.states is the authoritative record of what has happened so far in the game, and we don't want to mess it up with these speculative moves from the simulations.

Now we need to start keeping statistics on the game states that the AI hits during each run of run_simulation. The AI should pick the first unknown game state it reaches to add to the tables.

```
class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        self.wins = {}
        self.plays = {}

    # ...

    def run_simulation(self):
        visited_states = set()
        states_copy = self.states[:]
        state = states_copy[-1]
        player = self.board.current_
player(state)

        expand = True
        for t in xrange(self.max_moves):
            legal = self.board.legal_
plays(states_copy)

            play = choice(legal)
            state = self.board.next_state(state,
play)
            states_copy.append(state)

            # `player` here and below refers to
            # the player who moved into that
            # particular state.
            if expand and (player, state) not in
self.plays:
                expand = False
                self.plays[(player, state)] = 0
                self.wins[(player, state)] = 0

            visited_states.add((player, state))

            player = self.board.current_
player(state)
            winner = self.board.winner(states_
copy)
            if winner:
                break

        for player, state in visited_states:
            if (player, state) not in self.
plays:
                continue
            self.plays[(player, state)] += 1
            if player == winner:
                self.wins[(player, state)] += 1
```

Here we've added two dictionaries to the AI, wins and plays, which will contain the counts for every game state that is being tracked. The run_simulation method now checks to see if the current state is the first new one it has encountered this call, and, if not, adds the state to both plays and wins, setting both values to zero. This method also adds every game state that it goes through to a set, and at the end updates plays and wins with those states in the set that are in the plays and wins dicts. We are now ready to base the AI's final decision on these statistics.

```
from __future__ import division
# ...

class MonteCarlo(object):
    # ...

    def get_play(self):
        self.max_depth = 0
        state = self.states[-1]
        player = self.board.current_
player(state)
        legal = self.board.legal_plays(self.
states[:])

        # Bail out early if there is no real
        # choice to be made.
        if not legal:
            return
        if len(legal) == 1:
```

```
            return legal[0]

        games = 0
        begin = datetime.datetime.utcnow()
        while datetime.datetime.utcnow() - begin
< self.calculation_time:
            self.run_simulation()
            games += 1

        moves_states = [(p, self.board.next_
state(state, p)) for p in legal]

        # Display the number of calls of `run_
        # simulation` and the time elapsed.
        print games, datetime.datetime.utcnow()
- begin

        # Pick the move with the highest
        # percentage of wins.
        percent_wins, move = max(
            (self.wins.get((player, S), 0) /
             self.plays.get((player, S), 1),
             p)
            for p, S in moves_states
        )

        # Display the stats for each possible
        # play.
        for x in sorted(
            ((100 * self.wins.get((player, S),
0) /
              self.plays.get((player, S), 1),
              self.wins.get((player, S), 0),
              self.plays.get((player, S), 0), p)
             for p, S in moves_states),
            reverse=True
        ):
            print "{3}: {0:.2f}% ({1} / {2})".
format(*x)

        print "Maximum depth searched:", self.
max_depth

        return move
```

We have added three things in this step. First, we allow get_play to return early if there are no choices or only one choice to make. Next, we've added output of some debugging information, including the statistics for the possible moves this turn and an attribute that will keep track of the maximum depth searched in the selection phase of the playouts. Finally, we've added code that picks out the move with the highest win percentage out of the possible moves and returns it.

But we are not quite finished yet. Currently, our AI is using pure randomness for its playouts. We need to implement UCB1 for positions where the legal plays are all in the stats tables, so the next trial play is based on that information.

```
# ...
from math import log, sqrt

class MonteCarlo(object):
    def __init__(self, board, **kwargs):
        # ...
        self.C = kwargs.get('C', 1.4)

    # ...

    def run_simulation(self):
        # A bit of an optimization here, so we
        # have a local variable lookup instead
        # of an attribute access each loop.
        plays, wins = self.plays, self.wins

        visited_states = set()
        states_copy = self.states[:]
        state = states_copy[-1]
        player = self.board.current_
player(state)

        expand = True
        for t in xrange(1, self.max_moves + 1):
            legal = self.board.legal_
plays(states_copy)
            moves_states = [(p, self.board.next_
state(state, p)) for p in legal]

            if all(plays.get((player, S)) for p,
S in moves_states):
                # If we have stats on all of the
                # legal moves here, use them.
                log_total = log(
                    sum(plays[(player, S)] for
p, S in moves_states))
                value, move, state = max(
                    ((wins[(player, S)] /
```

```
plays[(player, S)]) +
                    self.C * sqrt(log_total /
plays[(player, S)]), p, S)
                    for p, S in moves_states
                )
            else:
                # Otherwise, just make an
                # arbitrary decision.
                move, state = choice(moves_
states)

            states_copy.append(state)

            # `player` here and below refers to
            # the player who moved into that
            # particular state.
            if expand and (player, state) not in
plays:
                expand = False
                plays[(player, state)] = 0
                wins[(player, state)] = 0
                if t > self.max_depth:
                    self.max_depth = t

            visited_states.add((player, state))

            player = self.board.current_
player(state)
            winner = self.board.winner(states_
copy)
            if winner:
                break

        for player, state in visited_states:
            if (player, state) not in plays:
                continue
            plays[(player, state)] += 1
            if player == winner:
                wins[(player, state)] += 1
```

The main addition here is the check to see if all of the results of the legal moves are in the plays dictionary. If they aren't available, it defaults to the original random choice. But if the statistics are all available, the move with the highest value according to the confidence interval formula is chosen. This formula adds together two parts. The first part is just the win ratio, but the second part is a term that grows slowly as a particular move remains neglected. Eventually, if a node with a poor win rate is neglected long enough, it will begin to be chosen again. This term can be tweaked using the configuration parameter C added to__init__ above. Larger values of C will encourage more exploration of the possibilities, and smaller values will cause the AI to prefer concentrating on known good moves. Also note that the self.max_depth attribute from the previous code block is now updated when a new node is added and its depth exceeds the previous self.max_depth.

So there we have it. If there are no mistakes, you should now have an AI that will make reasonable decisions for a variety of board games. I've left a suitable implementation of Board as an exercise for the reader, but one thing I've left out here is a way of actually allowing a user to play against the AI. A toy framework for this can be found at *github.com/jbradberry/boardgame-socketserver* and *github.com/jbradberry/boardgame-socketplayer*. ■

Jeff is a software engineer specializing in Python. He currently works for Caktus Group [caktusgroup.com], a Django consulting firm based in Durham, North Carolina. He has a degree in Applied Mathematics, and occasionally gets the itch to mix math with his programming.

Reprinted with permission of the original author.
First appeared in *hn.my/montecarlo* (jeffbradberry.com)

Will Your Hardware Startup Make Money?

Not for the first 5,000 units, but that's okay.

By BEN EINSTEIN

We talk to lots of founders who underestimate how hard it is to make money selling consumer hardware, especially on their first production run. If your product costs $30 to produce, and you sell it for $99, you're turning a profit, right?

Not so fast.

I'll lay it out for you. First, let's manufacture a fictional pair of bluetooth headphones, the Bolt-o-Phones. We need to make a few assumptions:

- Our Bolt-o-Phones will be sold for $99 MSRP (the manufacturer's suggested retail price)
- Our first production run will be 5,000 units
- Product development will take 9 months
- A small, 5-person team will work full-time on shipping this product

Getting Started

Most companies spend extensive time — and money — on product development. Simple products cost $100k–500k to develop, and they usually take roughly 6–9 months. More complex products can cost millions and take years.

In order to prepare our Bolt-o-Phones for the manufacturing process, we need to hire mechanical and electrical engineers, an industrial designer and an operations person. These employees will spend 9 months talking to users, building prototypes and getting ready to manufacture the product. Our costs will look something like this:

Bill of Materials is Just the Beginning

Once product development is finished, we'll have a final list of parts used to make our headphones (called a Bill of Materials, or BOM for short). This is the most fundamental cost structure we have to deal with as a hardware company. We can't raise money from investors or launch a crowdfunding campaign until we have a solid understanding of BOM costs.

The BOM includes all plastic parts we need molded, printed circuit board and other components we need to buy, glue to assemble the plastics, and the packaging in

Product Dev Cost	Description	Rate	Cost w/ Overhead	Amortized Cost
CEO/founder/operations	Management, investors, manfuacturing	$60k/yr	$56,250	$11.25
Mechanical Engineer	Design and prototyping of plastics	$80k/yr	$75,000	$15.00
Electrical Engineer	Design and debug of PCB	$80k/yr	$75,000	$15.00
Embedded Systems	Design and debug of firmware	$80k/yr	$75,000	$15.00
Industrial Design	Plastics design, color, texture, feel	$60k/yr	$56,250	$11.25
Prototypes/tools	3D printing, PCB fab/assembly, finishing, rework	$20k	$20,000	$4.00
Product Development Cost (one revision at 5,000 units)			**$357,500**	**$71.50**

This estimate is highly variable depending on product complexity, team makeup, etc. It will impact profitability more than any other cost.

which the Bolt-o-Phones are sold. Each part is laid out on a table with all the information required to make a single pair of Bolt-o-Phones: part number, quantity per unit, vendor, lead times, costs and various notes.

Part Number	Description	Manufacturer/Supplier	Lead Time (wk)	Qty	Part Cost	Extended Cost	Notes
Molded Parts							
BTHEAD.PP.001	Ear Cup Base (left)	Dongguan Tool Co	8	1	$0.18	$0.18	CM tools, debugs, and shoots
BTHEAD.PP.002	Ear Cup Base (right)	Dongguan Tool Co	8	1	$0.18	$0.18	CM tools, debugs, and shoots
BTHEAD.PP.003	Band Substrate	Dongguan Tool Co	8	1	$0.68	$0.68	CM tools, debugs, and shoots
BTHEAD.PP.004	Band Rubber	Dongguan Tool Co	8	1	$0.54	$0.54	CM tools, debugs, and shoots
BTHEAD.PP.005	Band Underside	Dongguan Tool Co	8	1	$0.22	$0.22	CM tools, debugs, and shoots
BTHEAD.PP.006	Battery Door	Dongguan Tool Co	8	1	$0.22	$0.22	CM tools, debugs, and shoots
Stamped Parts							
BTHEAD.ST.001	Band Spring	Shenzhen Stamping Ltd	10	1	$0.83	$0.83	Stamped, treated and delivered to CM
BTHEAD.ST.002	Ear Cup Cover	Shenzhen Stamping Ltd	10	2	$1.01	$2.02	Stamped, finished and delivered to CM
Die-cast Parts							
None							
Purchased Parts							
BTHEAD.OTS.001	Speaker Driver	Kinmore	2	2	$1.10	$2.20	Consigned
BTHEAD.OTS.002	Accessory Cable	Monster	1	1	$0.50	$0.50	Consigned
BTHEAD.OTS.003	Lythium Polymer Battery	Pannasonic	4	2	$1.22	$2.44	Consigned
BTHEAD.OTS.004	Headphone Bag	Shenzhen Fabrics Ltd	8	1	$0.85	$0.85	Consigned
BTHEAD.OTS.005	Screw, stainless PP	Dongguan Screw Co	1	8	$0.02	$0.16	Purchased by CM
BTHEAD.OTS.006	Glue	3M	1	0.05	$18.00	$0.90	Purchased by CM
Electronics							
BTHEAD.PCB.001	PCB Main	Dongguan Tool Co	4	1	$9.49	$9.49	PCB board assembled/flashed w/ components
BTHEAD.PCB.002	PCB Daughter	Dongguan Tool Co	4	1	$4.22	$4.22	PCB board assembled w/ components
Packaging							
BTHEAD.PKG.001	Master Carton	Dongguan Tool Co	1	0.050	$2.05	$0.10	1 master carton holds 20 units, cardboard + 1 color
BTHEAD.PKG.002	Gift Box	Dongguan Tool Co	2	1	$3.58	$3.58	Box for single unit for retail
BTHEAD.PKG.003	Gift Box Insert Top	Dongguan Tool Co	4	1	$1.45	$1.45	Internal packaging
BTHEAD.PKG.004	Gift Box Insert Bottom	Dongguan Tool Co	4	1	$1.40	$1.40	Internal packaging
BOM (at 5,000 units)						**$32.16**	

Don't Forget About COGS

The BOM leaves out some critical costs associated with each unit. Each pair of Bolt-o-Phones takes time for workers in China to assemble. And then we have to ship the product all over the world. These costs, and others, are reflected in the Cost of Goods Sold (commonly known as COGS). Financially, COGS are calculated using inventory costs, but for startups it's easier to think of COGS as an 'extended BOM.' I'll include anything we pay for on a per unit basis that we can't order more of, but that is required to get the product out the door (such as duties, scrap rates, and return rates, all of which are calculated as a percentage of the BOM cost):

Process/Fees	Description	Rate	Extended Cost
		BOM Cost	***$32.16***
Assembly Labor	Time for assembly in China	10 minutes @ $5.00/hr	$0.83
CM Profit Margin	Margin CM makes on BOM cost	5% of BOM cost	$1.61
Scrap Rate	Units rejected for QC	5% of BOM cost	$1.61
Duties	Import and export fees/taxes	5% of BOM cost	$1.61
Freight	Freight from Hong Kong to US	$1/unit sea from HK, $3/unit land in US	$4.00
3PL	Thrid party logistics (warehousing and pallet pick)	7% of BOM cost	$2.25
Returns	Returns and reverse logistics	5% return rate + $20/unit returned logistics	$3.20
COGS (at 5,000 units)			**$47.28**

Everyone Has Fixed Costs

In addition, we have fixed costs associated with our first production run. When the company is still young, and has yet to turn a substantial profit, these fixed costs make a significant impact on our financials. Fixed costs are things we pay for once for every design, like tooling for plastic parts, FCC fees for bluetooth radio certification, UL/CE product certification costs or value-added services from the CM. It's best to outline these costs independently, as they can be significant expenses, but it's also helpful to show the amortized cost over the production run of 5,000 units, which is what I've done here:

Fixed Cost	Description	Rate	Cost	Amortized Cost
		COGS	***$236,376***	***$47.28***
CM NRE	Contract manufacturing NRE, translation, setup, testing	100hrs @ $50/hr + $20,000 fixed	$25,000	$5.00
EVT and DVT builds	Unsellable units for debugging	500 units @ BOM cost	$16,081	$3.22
Injection Molding Tools	Design, production, setup, debug	6 tools @ $5,000 ea	$30,000	$6.00
Stamping Tools	Design, production, setup, debug	2 tools @ $2,500 ea	$5,000	$1.00
FCC Bluetooth Certification	FCC-ID embedded active transmitter cert	$5,000/PCB revision	$5,000	$1.00
UL Certification	UL/CE is often required for retail sales	$15,000/SKU	$15,000	$3.00
COGS + Fixed (at 5,000 units)			**$332,457**	**$66.49**

After 9 months of development, manufacturing and logistics, we wind up with 5,000 units of our product sitting in a warehouse somewhere in the US. We've spent around $690k ($360k for development and $330k for manufacturing) to get here, and we are ready to send our customers their gorgeous Bolt-o-Phones.

Go Direct First

Originally, a BOM cost of $32.16 would imply that we can make money selling the product for $99 anywhere we want. But once all the other expenses are factored into the equation, our distribution options diminish significantly. The three typical distribution options companies have at their disposal are as follows:

- Direct (sold through your own website, where no margin is paid but you must process payments and pay for fulfillment)
- Online retail/e-tail (a third-party seller with no physical store, and that takes low margins)
- Traditional physical retail (a physical store with a distribution network and standard retail margins)

Our profit on each unit varies hugely depending on the channel into which we sell:

Distribution Costs	Direct (our website/crowdfunding)	Online retail (Amazon/NewEgg)	Physical Retail (Best Buy/Apple)
Margin (off MSRP)	0%	18%	40%
Gross Unit Sale Price	$99.00	$81.18	$59.40
Processing Fee	-$4.95	-	-
Order fullfillment	-$2.36	-	-
Fully-loaded Unit Cost	-$66.49	-$66.49	-$66.49
Insurance	-	-$3.00	-$3.00
Display/Placement Cost	-	-	-$2.00
Load-in Cost	-	-	-$0.50
Gross Profit/Loss Per Unit	**$25.19**	**$11.69**	**-$12.59**
Gross Profit/Loss	**$125,974**	**$58,443**	**-$62,957**
Net Profit/Loss	**-$231,526**	**-$299,057**	**-$420,457**

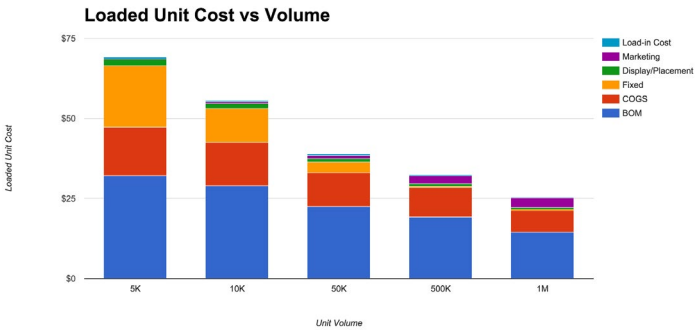
Notice that each unit sold via physical retail actually LOSES money. This is why it's very difficult for a small company with a limited amount of cash to go straight to retail distribution on their first production run.

Trends at Scale

A 5,000-unit production run may be pretty daunting for first-time founders, but it's peanuts compared to successful consumer products. Real success comes from selling lots of units, mainly due to massive economies of scale. At high volumes:

- Amortized fixed costs go to zero, due to the high number of units that are produced
- Negotiation leverage increases with retailers for better margins. Retailers care about "walk-in value" (in other words, how likely a customer is to walk into a store for your product), and as your product becomes more popular/well-known, your walk-in value increases.
- Negotiation leverage increases with suppliers for better prices
- CMs extend large lines of credit to good customers, allowing you to pay for your product after it's made (sometimes 90 or 120 days after, which significantly reduces cashflow problems.)
- Scrap and return rates go down as manufacturing tolerances tighten and customer support improves

The fully-loaded unit cost of Bolt-o-Phones will change dramatically as we manufacture more and more units. Notice how most of our costs decrease substantially, aside from marketing, which tends to increase over time:



The profitability of the company also drastically changes, which is driven by increased leverage from margin negotiations and lower unit costs:

Cost	5k Units	10k Units	50k Units	500k Units	1M Units
Margin (off MSRP)	40%	37.5%	35%	32%	30%
Gross Unit Sale Price	$59.40	$61.88	$64.35	$67.32	$69.30
Fully-loaded Unit Cost	-$66.49	-$59.84	-$46.54	-$39.89	-$29.92
Insurance	-$3.00	-$1.50	-$0.30	-$0.003	-$0.003
Display/Placement Cost	-$2.00	-$1.50	-$1.00	-$0.75	-$0.60
Marketing/Sales	-$0.10	-$0.50	-$1.00	-$2.50	-$3.00
Load-in Cost	-$0.50	-$0.45	-$0.40	-$0.25	-$0.10
Gross Profit/Loss Per Unit	**-$12.69**	**-$1.92**	**$15.11**	**$23.92**	**$35.68**
Gross Profit/Loss	**-$63,457**	**-$19,173**	**$755,301**	**$11,961,076**	**$35,675,864**

Although shipping 5,000 units of anything is an amazing accomplishment, this table illustrates just how difficult it is to make money in the early days of your hardware business. Even the most successful crowdfunding campaigns (think Canary, Pebble, Oculus, Ouya, etc.) struggled to make money on their first production run. It takes a massive manufacturing scale like Fitbit (with 10.9 million units sold in 2014) to build a venture-scale, profitable business (Fitbit is currently worth around $9B.) However, don't be discouraged! Selling 10.9M units seemed like a pipe dream to James and Eric when they started Fitbit in 2007. ■

Ben Einstein is a founder and partner at Bolt. A product vision and prototyping expert, Ben is instrumental in bringing many products to market ranging from consumer electronics to clean energy for everything from Fortune 500 companies to small startups.

How I Came To Find Linux

BY IAN MURDOCK

I SAW MY FIRST Sun workstation in the winter of 1992, when I was an undergraduate at Purdue University. At the time, I was a student in the Krannert School of Management, and a childhood love of computers had just been reawakened by a mandatory computer programming course I had taken during the fall semester. (We were given the choice between COBOL and FORTRAN — which even in 1992 seemed highly dated — and I had picked COBOL because it seemed the more "business" of the two.)

Ten years or so earlier, my father, a professor of entomology at Purdue, had replaced his typewriter at work with an Apple II+. Thinking his nine-year-old son might get a kick out of it, he brought it home one weekend along with a Space Invaders-like game he had bought at the local ComputerLand. I spent hours on the computer that weekend. Before long, I was accompanying Dad to the lab at every opportunity so I could spend as much time on the computer as possible.

Being a nine-year-old boy, I was, predictably, attracted by the games at first, and my interest in games led to my first exposure to programming: computer magazines that included code listings for very simple games, which I would laboriously key in to the Apple — and, after hours of toil, hope that I hadn't made a mistake. (The Apple II, at least out of the box, utilized a simple line editor, so going back and making changes was very tedious, not to mention finding the errors in the first place.)

Not long after, I met Lee Sudlow while hanging around the lab on weekends. Lee was one of Dad's graduate students and he had begun to use the Apple to assist in his experiments. Lee was always happy to explain what he was doing as I hovered over his shoulder watching, his helpfulness no doubt motivated — at least in part — by the fact that the snot-nosed nine-year-old scrutinizing his every move was his faculty advisor's son. Oblivious to such things, I watched with fascination as he punched code into the Apple — code that he thought up himself, not code that he was reading from a computer magazine.

Between learning by example through studying the code in the magazines and Lee's occasional tutelage, I was writing games and other simple programs before long, first in Applesoft BASIC and, later, in 6502 assembly language. To encourage my growing interest, Dad eventually bought an Apple IIe for home, and my love affair with the computer continued for several more years. However, as I entered my teenage years, the computer was gradually replaced with more pressing things, like baseball, music, and girls, and by the mid-1980s, the Apple was gathering dust in my bedroom closet alongside my collection of Hardy Boys novels and Star Wars action figures.

My obsession with the computer lay dormant for the next half-dozen years until it was fortuitously reactivated during that COBOL course in the fall of 1992. When the course ended, I naturally lost my account on the IBM 3090 mainframe where we did our assignments and lab work. Fortunately, as a student, I was entitled to a personal account on one of the university computing center's machines, either the IBM or one of three Sequent Symmetry minicomputers running DYNIX, a variant of the UNIX operating system. A friend convinced me that UNIX was more interesting and had a brighter future than IBM's VM/CMS, and I took his advice and applied for an account on one of the Sequent machines. The following week, I was the proud owner

> **“I bought a box of thirty floppy diskettes and began the slow process of downloading Linux to the floppies from a PC lab.”**

of an account on sage.cc, complete with the princely allocation of 500 kilobytes of disk storage. (Yes, I’m being sarcastic — 500 kilobytes was a miserly sum, even for 1992. I eventually found ways to circumvent it.)

My appetite for UNIX was ravenous that winter. I spent most evenings in the basement of the MATH building basking in the green phosphorescent glow of the Z-29 terminals, exploring every nook and cranny of the UNIX system upstairs. It was eerily quiet in those terminal rooms, the only sound being the clack clack clack of a few dozen keyboards and the occasional whisper of, “Hey, look at this….” Often, after an evening of exploration, I would exit the building the long way, walking past the plate-glass window where the computing center housed its machines, gazing in awe at the refrigerator-sized Sequent Symmetry I had just been using, watching the blinking lights and knowing that hundreds of people were still inside, if only virtually, thanks to the magic of time-sharing, a technique advanced computers used to divide the machine’s computational power among many users, providing the illusion that each user was the only one. Above all, I looked with envy at the system operators privileged enough to sit on the other side of that plate-glass window wielding the almighty power of the “superuser” at the system console.

Unsatisfied with the Z-29s, I began prowling around campus after dark with a friend, Jason Balicki, to see what else could be found. Jason had been in the computer science program for a few years, so he knew where to look (though we did our share of new exploration — that was part of the fun — entering buildings at night and trying the doorknobs of rooms that looked like they might hold computers to see if they were unlocked).

The best labs, I learned, were in the engineering administration building (referred to around campus by its unfortunate acronym, ENAD), where several rooms of X terminals offered a grayscale graphical interface to the Sequent and other UNIX machines around campus. Soon, my preferred “hacking” spot (a term Jason had introduced to me) was in one of the X terminal labs, which were technically only for engineering students, a restriction that was not enforced by passwords — and that we dutifully ignored.

But the mother lode of the ENAD building was to be found in its labs of Sun workstations. Unlike the lowly Z-29s and even the comparatively advanced X terminals, the Suns were things of beauty, with sleek cases and high-resolution color displays. Furthermore, Jason explained that they ran the best UNIX there was, SunOS, though the Suns were considerably better locked down than the X terminals, requiring an account on the engineering computer network to access them, so I didn’t get a chance to actually get my hands on SunOS until much later.

I was also accessing UNIX from home via my Intel 80286-based PC and a 2400-baud modem, which saved me the trek across campus to the computer lab on particularly cold days. Being able to get to the Sequent from home was great, but I wanted to replicate the experience of the ENAD building’s X terminals, so one day, in January 1993, I set out to find an X server that would run on my PC. As I searched for such a thing on Usenet, I stumbled across something called “Linux.”

Linux wasn't an X server, of course, but it was something much better: A complete UNIX-alike operating system for PCs, something I hadn't even contemplated could exist. Unfortunately, it required a 386 processor or better, and my PC only had a 286. So, I began to save my pennies for a machine fast enough to run it, and while I did that, I devoured everything I could get my hands on about the object of my desire. A few weeks later, I posted a message to Purdue's computing interest Usenet group asking if anyone on campus was running Linux — and got one response, from a computer science student named Mike Dickey, who happily invited me over to show me his Linux setup. Inspired, I bought a box of thirty floppy diskettes and began the slow process of downloading Linux to the floppies from a PC lab in the Krannert building, though it would be another month before I could afford an actual computer on which to install it. Finally, I could wait no longer, and Jason and I found an unlocked computer lab in one of the dorms containing a single PC, and in the middle of the night one evening in February, we proceeded to install Linux on that lab PC. I still occasionally wonder what the unfortunate student first to the lab the next morning must have thought.

Linux had been created about a year and a half before by Linus Torvalds, a twenty-one-year-old computer science undergraduate at Helsinki University. A longtime computer enthusiast, Torvalds had followed a path roughly similar to my own, though he began his programming career on a Commodore Vic-20, and he hadn't gotten distracted by the more traditional interests of teenage boys as the '80s progressed. Torvalds' first exposure to UNIX was in 1990 during a course at the university and, like me, it had been love at first sight.

In the fall of that same year, Torvalds took a course in operating systems that used the textbook Operating Systems: Design and Implementation by Andrew Tanenbaum, a professor of computer science at Amsterdam's Vrije Universiteit. Tanenbaum's book taught operating systems by example through a UNIX clone for PCs he had written called MINIX, and his book included the complete source code — the human readable (and editable) programming code — for MINIX along with a set of floppy diskettes so that readers could actually install, use, and modify the operating system.

Intrigued, Torvalds bought a PC in early 1991 and joined the burgeoning MINIX community, tens of thousands strong and largely held together by the Usenet newsgroup comp.os.minix. He began experimenting not only with MINIX but also with the new task-switching capabilities of the Intel 80386 processor that powered his PC. (Task-switching makes it easier to run more than one program on the processor at the same time, one of the requirements of a time-sharing system like the Sequent Symmetry I would discover the following year at Purdue.) By the summer of 1991, Torvalds' experiments with task-switching were beginning to evolve into a full-blown operating system kernel, the basic piece of software in an operating system that mediates access to the CPU, memory, disks and other devices in the computer and provides a simpler interface to these basic computing functions that allows complex applications to be written more easily.

MINIX was not the only "hobbyist-friendly" operating system project that existed in 1991, though it was one of only a handful that was complete enough to be usable, and one of only a few that would run on the lowly PC. The best-known operating system project by far was GNU, presided over by Richard Stallman. Stallman, who had been programming since the mid-1960s and had been a systems programmer at MIT from 1971 to 1983, was an old-school "hacker," someone who engages in computing for its own sake and believes, militantly in some cases (including Stallman's), that all information should be freely shared.

The GNU project's goal was to produce a free operating system (free not only in price, but also free in the sense that it could be freely modified) that was compatible with UNIX (GNU was a so-called recursive acronym for "GNU's Not UNIX," so-called because it employed a powerful technique often used by programmers called recursion that involves a computation using itself as one of its inputs). Stallman launched the GNU project in 1983 in response to the growing market for proprietary software — software for which the source code could not be modified and was often not even available.

Proprietary software was a fairly new development in the early 1980s and, to Stallman, a very disturbing one. Up to that point, software had largely been distributed freely with hardware, and hackers often shared copies of its source code along with their own changes

and improvements. Stallman considered the growing trend toward proprietary software nothing short of the first step toward a digital 1984 in which computer users, and eventually all of society, would be held captive by greedy corporate interests, and he was determined to stop it.

By mid-1991, Stallman and a loosely-knit group of volunteers had assembled most of the GNU operating system — a compiler, a debugger, an editor, a command interpreter (or "shell"), and a variety of utilities and programming libraries that were just like UNIX, only better — the GNU versions were almost universally held to be superior to their namesakes. The only piece that was missing was the kernel, and a small team had just been created at Stallman's Free Software Foundation, a non-profit organization he had formed in 1985 to oversee development of GNU and serve as a guardian of sorts for free software, to write that final piece. Hackers around the world believed it would just be a matter of time until GNU was finished and available, and they would finally have an operating system free of corporate encumbrances.

Half a world away, Torvalds' own operating system kernel was becoming complete enough to release to the world. In a now-famous Usenet posting to comp.os.minix on August 25, 1991, he wrote:

> *Hello everybody out there using minix –*
>
> *I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).*

The response was immediate and overwhelming. While everyone expected GNU to be done imminently, it was not available yet, at least not in a form that could be used without a UNIX scaffolding underneath. And while MINIX was popular, it was not free, though it was certainly inexpensive compared to the other UNIXes. Perhaps most importantly, though, MINIX was intended primarily as a teaching aid, not production software, so Tanenbaum was loathe to include many of the patches, or changes to the operating system, that extended its capabilities which flowed in daily from hordes of enthusiastic users around the world, fearing their addition would make MINIX too complicated and, thus, harder for his students to understand.

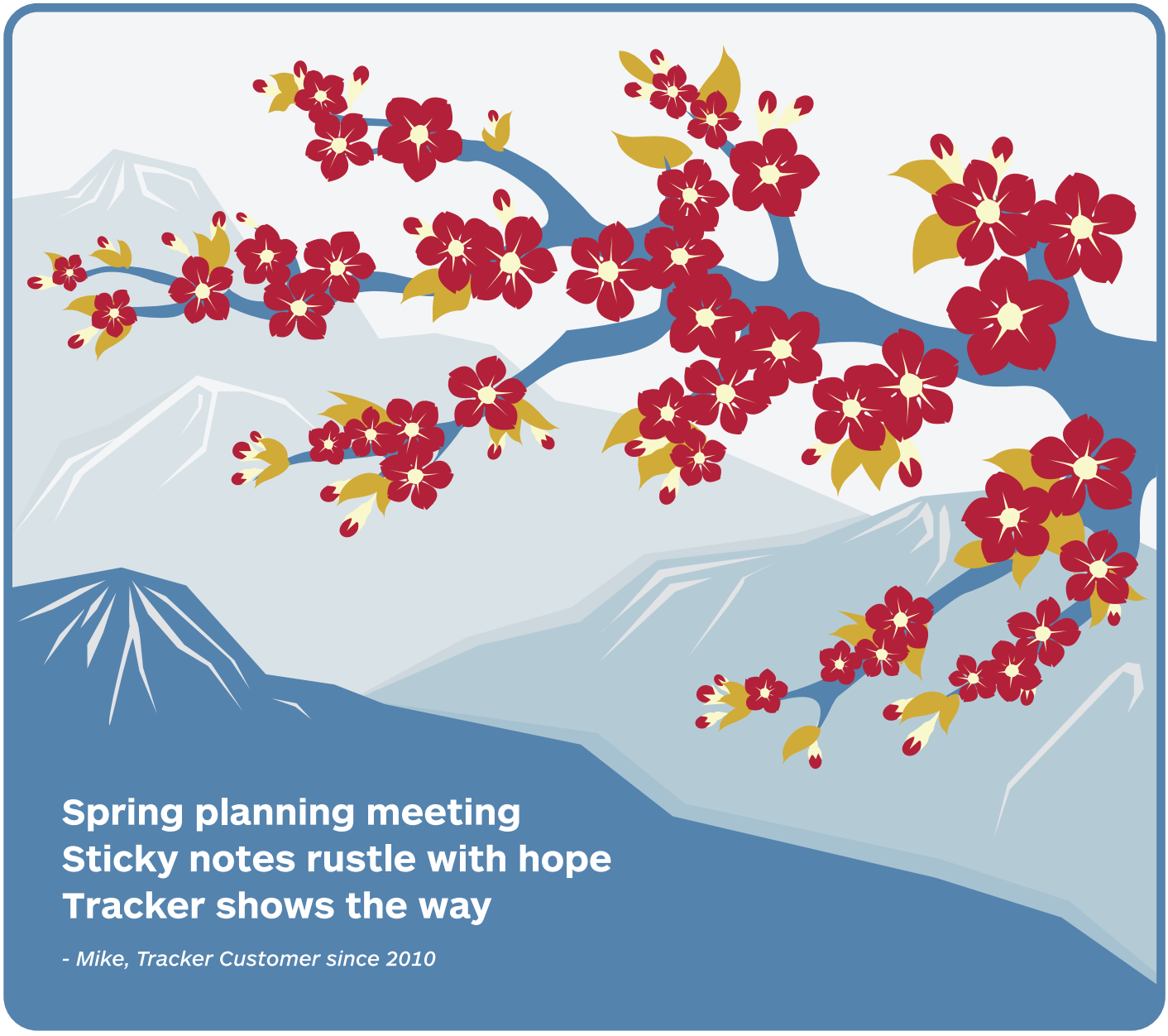
The lure of a UNIX-like operating system for PCs, no matter how imperfect, that was free and could evolve at the speed its community wanted it to evolve was too much for many MINIX users to resist, and they began flocking in droves to Torvalds' new OS, which in the fall of 1991 would be dubbed "Linux." But Linux was just a kernel — it required a variety of tools and applications be installed on top of it to make it actually do anything useful. Fortunately, most of these already existed thanks to Stallman's GNU project.

By 1992, a few intrepid users began to assemble sets of floppy diskette images that combined Linux with the GNU software tool chain to make it easier for new users to get up and running. These collections (later called "distributions") got progressively better, and by the time I finally got my PC in March of 1993, the Softlanding Linux System (or SLS) distribution had expanded to those thirty diskettes and now included a wealth of applications — and, yes, the very same software that powered the X terminals in the ENAD building.

I never did get around to trying to connect the Linux-based X server now on my PC to the Sequent, which would have been painfully slow at 2400 baud — several thousand times slower than the speeds of today. Now I had my very own UNIX to explore right there on my desk. And explore I did, in a veritable UNIX crash course. Once I got over the thrill of being the "superuser," the unspeakable power I had previously seen only behind plate glass, I became enraptured not so much by Linux itself as by the process in which it had been created — hundreds of people hacking away at their own little corner of the system and using the Internet to swap code, slowly but surely making the system better with each change — and set out to make my own contribution to the growing community, a new distribution called Debian that would be easier to use and more robust because it would be built and maintained collaboratively by its users, much like Linux. ■

A longtime Linux user, developer, and advocate, Ian Murdock founded the Debian project in 1993. Today, Debian is one of the most popular Linux distributions in the world, with millions of users worldwide. Ian has also held positions with the Linux Foundation, Sun Microsystems, and Salesforce.

Reprinted with permission of the original author. First appeared in *hn.my/debian* (ianmurdock.com)



Discover the newly redesigned Pivotal Tracker

As our customers know too well, building software is challenging. That's why we created Pivotal Tracker, a pleasure-to-use project management tool, designed to facilitate constructive communication, keep teams focused, and reflect the true status of all your software projects.

With a new UI, cross-project funcionality, in-app notifications and more, staying zen in the face of looming business deadlines just got a little easier.

Sign up for a free trial, no credit card required, at pivotaltracker.com.

PivotalTracker

Build better software, faster.