goxpyriment — Getting Started

Christophe Pallier christophe@pallier.org

2026-04-06

Contents

Getting Started with goxpyriment

Welcome! If you are a psychologist or neuroscientist used to building experiments in Python (with **Expyriment** or **PsychoPy**), you are in the right place.

`goxpyriment` brings the high-level simplicity of those tools to the **Go** programming language, offering significant advantages in timing precision and ease of sharing your work.

Why Go? (and why goxpyriment?)

If you've ever spent three hours fixing a `conda` environment or `pip` conflict just to run a simple experiment on a lab computer, you'll love Go.

1. **Zero-Dependency Deployment**: When you build a Go experiment, it produces a **single binary** (an `.exe` on Windows, an AppImage on Linux, a `.app` on macOS). Drop it on any lab computer and it just works — no Python installation required.
2. **Timing Precision**: Go is a compiled language with a very efficient runtime. `goxpyriment` runs the stimulus loop VSYNC-locked with GC pauses disabled, giving you sub-millisecond frame jitter on typical hardware.
3. **AI-friendly API**: The linear, consistent API makes it very well suited to "vibe-coding" — describe your paradigm in plain language to Claude, Gemini, or ChatGPT and the generated code is usually 90 % ready to run immediately.

Mapping Concepts: Python to Go

Expyriment (Python)	goxpyriment (Go)	Note
`exp = design.Experiment(...)`	`exp := control.NewExperimentFromFlags(...)`	The central manager (flag-aware).
`exp.initialize()`	(handled by `NewExperimentFromFlags`)	SDL, audio and font all initialized.
`stim.present()`	`exp.Show(stim)`	Clear → draw → flip.
`exp.clock.wait(1000)`	`exp.Wait(1000)`	OS-responsive wait; aborts on ESC.
`key, rt = exp.keyboard.wait()`	`key, rt, err := exp.Keyboard.WaitKeysRT(keys, timeout)`	Response + reaction time.

To run the following tutorials, you have a Go developement environement. See here for installation instructions.

Tutorial 1: Your First Trial

A classic sequence: **Fixation Cross** (500 ms) → **Target Circle** → **Wait for Spacebar**.

```go
package main

import (
    "log"

    "github.com/chrplr/goxpyriment/control"
    "github.com/chrplr/goxpyriment/stimuli"
)

func main() {
    exp := control.NewExperimentFromFlags("SimpleTrial", control.Black, control.White,
    defer exp.End()

    fixation := stimuli.NewFixCross(20, 3, control.White)
    target   := stimuli.NewCircle(50, control.White)

    exp.ShowInstructions("Press SPACE when you see the circle.")

    exp.Show(fixation)
    exp.Wait(500)          // hold fixation for 500 ms

    exp.Show(target)
```

```
    exp.Keyboard.WaitKey(control.K_SPACE)
}
```

Tutorial 2: Blocks, Trials, and Data Logging

For real research you need multiple trials and a CSV result file.

```
package main

import (
    "fmt"
    "log"

    "github.com/chrplr/goxpyriment/control"
    "github.com/chrplr/goxpyriment/design"
    "github.com/chrplr/goxpyriment/stimuli"
)

func main() {
    exp := control.NewExperimentFromFlags("ParityTask", control.Black, control.White, 3
    defer exp.End()

    exp.AddDataVariableNames([]string{"number", "is_even", "rt_ms", "correct"})

    err := exp.Run(func() error {
        exp.ShowInstructions(
            "Is the number EVEN or ODD?\n\nPress F for Even, J for Odd.",
        )

        numbers := []int{1, 2, 3, 4, 5, 6, 7, 8}
        trials  := design.MakeMultipliedShuffledList(numbers, 4) // 32 trials, randomiz

        for _, n := range trials {
            exp.Blank(1000)

            stim := stimuli.NewTextLine(fmt.Sprint(n), 0, 0, control.White)
            exp.Show(stim)

            // WaitKeysRT returns the key pressed, the reaction time in ms, and any err
            key, rt, _ := exp.Keyboard.WaitKeysRT(
                []control.Keycode{control.K_F, control.K_J}, -1,
            )

            isEven := (n % 2 == 0)
            correct := (isEven && key == control.K_F) || (!isEven && key == control.K_J
```

```
            // Subject ID and timestamp are prepended automatically.
            exp.Data.Add(n, isEven, rt, correct)
        }
        return control.EndLoop
    })
    if err != nil && !control.IsEndLoop(err) {
        log.Fatalf("experiment error: %v", err)
    }
}
```

exp.Run handles the detection of the ESC keypress to interrupt the experiment. The `.csv` result file (a CSV with a metadata header) is written to `~/goxpy_data/` automatically when the experiment ends. Each row gets the subject ID and a timestamp for free.

Tutorial 3: Rapid Serial Visual Presentation (RSVP Streams)

Many paradigms — RSVP, Attentional Blink, priming — need stimuli flashed at high speed with precise timing. `goxpyriment` has dedicated stream functions that:

- Disable GC during the stream (no garbage-collection pauses).
- Lock every onset and offset to a VSYNC boundary.
- Record a `TimingLog` (predicted vs. actual onset) for every stimulus.
- Capture any key or mouse event that occurs during the stream.

Text stream

The simplest entry point is `stimuli.PresentStreamOfText`:

```
package main

import (
    "fmt"
    "log"
    "time"

    "github.com/chrplr/goxpyriment/control"
    "github.com/chrplr/goxpyriment/stimuli"
)

func main() {
    exp := control.NewExperimentFromFlags("RSVP Demo", control.Black, control.White, 36
    defer exp.End()

    exp.AddDataVariableNames([]string{"target_pos", "detected", "rt_ms"})

    // A stream of words; "TIGER" is the target at position 3 (0-indexed).
```

```
    words     := []string{"CHAIR", "RIVER", "LAMP", "TIGER", "CLOCK", "STONE", "BREAD",
    targetPos := 3

    err := exp.Run(func() error {
        exp.ShowInstructions(
            "Words will flash rapidly on screen.\n\n" +
            "Press SPACE as quickly as possible when you see TIGER.\n\n" +
            "Press any key to start.",
        )

        // 200 ms on, 50 ms off per word → ~4 words per second
        on  := 200 * time.Millisecond
        off :=  50 * time.Millisecond

        events, logs, err := stimuli.PresentStreamOfText(
            exp.Screen, words, on, off,
            0, 0,          // centred on screen
            control.White,
        )
        if err != nil {
            return err
        }

        // Did the participant press SPACE during the stream?
        ev, detected := stimuli.FirstKeyPress(events, control.K_SPACE)
        rtMs := ev.Timestamp.Milliseconds()

        // Optional: inspect timing quality (jitter in ms per word).
        for i, l := range logs {
            fmt.Printf("word %d: target %d ms  actual %d ms  jitter %d ms\n",
                i, l.TargetOn.Milliseconds(), l.ActualOnset.Milliseconds(),
                (l.ActualOnset - l.TargetOn).Milliseconds())
        }

        exp.Data.Add(targetPos, detected, rtMs)
        return control.EndLoop
    })
    if err != nil && !control.IsEndLoop(err) {
        log.Fatalf("experiment error: %v", err)
    }
}
```

Image stream

For image or mixed stimulus streams, build the element list manually:

```
stims := []stimuli.VisualStimulus{pic1, pic2, fixation, pic3}
on  := 100 * time.Millisecond
off :=   0 * time.Millisecond

elements := stimuli.MakeRegularVisualStream(stims, on, off)
events, logs, err := stimuli.PresentStreamOfImages(exp.Screen, elements, 0, 0)
_ = events; _ = logs; _ = err  // inspect as shown in the text stream example
```

Audio stream

The same pattern exists for sounds:

```
tones     := []stimuli.AudioPlayable{tone440, tone880, tone440}
onsets    := []int{0, 500, 1000}   // ms from stream start
durations := []int{200, 200, 200}

elements, _ := stimuli.MakeSoundStream(tones, onsets, durations)
events, logs, err := stimuli.PlayStreamOfSounds(elements)
_ = events; _ = logs; _ = err
```

All three stream functions return `([]UserEvent, []TimingLog, error)`, making it straightforward to analyse timing quality and log participant responses.

Tutorial 4: Hardware-Precision RT with Event Timestamps

`WaitKeysRT` measures reaction time from the moment the function is called. That works well for a single-stimulus trial, but breaks down when several stimuli appear in sequence and you need RT from a specific onset.

Consider a **masked priming** paradigm: a prime word appears briefly, then a target appears 500 ms later, and you want RT measured from the prime onset — not from when `GetKeyEventTS` was called. With the standard approach you would need to record a timestamp before the prime and do arithmetic afterward. The event-timestamp API handles this directly.

SDL3 stamps every keyboard event with a hardware-interrupt time (`KeyboardEvent.Timestamp`, nanoseconds). `exp.ShowTS(stim)` returns the SDL nanosecond time captured immediately after the VSYNC flip. Because both values are on the same clock, their difference is hardware-precision RT — no arithmetic needed:

```
package main

import (
    "fmt"
    "log"

    "github.com/chrplr/goxpyriment/control"
    "github.com/chrplr/goxpyriment/stimuli"
```

```
)

func main() {
    exp := control.NewExperimentFromFlags("PrimingRT", control.Black, control.White, 36
    defer exp.End()

    exp.AddDataVariableNames([]string{"prime", "target", "key", "rt_prime_ns", "rt_targ

    primes  := []string{"DOCTOR", "NURSE", "BREAD", "TABLE"}
    targets := []string{"NURSE", "DOCTOR", "TABLE", "BREAD"}  // related / unrelated pa
    responseKeys := []control.Keycode{control.K_F, control.K_J}

    err := exp.Run(func() error {
        exp.ShowInstructions(
            "A word will flash, then a second word will appear.\n\n" +
            "F = Living thing   J = Non-living thing\n\n" +
            "Respond to the SECOND word as quickly as possible.\n\n" +
            "Press SPACE to start.",
        )

        for i := range primes {
            exp.Blank(500)   // inter-trial interval

            // 1. Show prime — record its VSYNC flip timestamp
            prime := stimuli.NewTextLine(primes[i], 0, 0, control.Gray)
            primeOnset, _ := exp.ShowTS(prime)
            prime.Unload()

            exp.Wait(500)  // prime–target SOA

            // 2. Show target — record its VSYNC flip timestamp
            target := stimuli.NewTextLine(targets[i], 0, 0, control.White)
            targetOnset, _ := exp.ShowTS(target)
            target.Unload()

            // 3. Wait for response — get hardware event timestamp
            key, eventTS, _ := exp.Keyboard.GetKeyEventTS(responseKeys, 3000)

            // 4. Compute RTs from each stimulus onset
            rtFromPrime  := int64(eventTS - primeOnset)   // nanoseconds
            rtFromTarget := int64(eventTS - targetOnset)  // nanoseconds

            exp.Data.Add(primes[i], targets[i], key, rtFromPrime, rtFromTarget)
            fmt.Printf("prime RT: %.1f ms   target RT: %.1f ms\n",
                float64(rtFromPrime)/1e6, float64(rtFromTarget)/1e6)
        }
        return control.EndLoop
```

```
    })
    if err != nil && !control.IsEndLoop(err) {
        log.Fatalf("experiment error: %v", err)
    }
}
```

Key observations:

- `exp.ShowTS(stim)` is a drop-in replacement for `exp.Show(stim)` — it does the same clear → draw → flip, and additionally returns the nanosecond timestamp of the flip.
- `GetKeyEventTS` returns the SDL3 event timestamp (not a polling delta), so subtracting any previously recorded `ShowTS` onset gives a physically meaningful RT.
- Both timestamps are in SDL nanoseconds (divide by `1e6` for milliseconds). Storing raw nanoseconds in the data file and converting offline is the recommended practice.
- `GetPressEventTS` provides the same capability for mouse responses.

Next Steps

- Browse the `examples/` folder for complete, documented paradigms (Stroop, Simon, Posner, Sperling, QUEST threshold estimation, and more).
- Read the User Manual for a deeper explanation of every concept, or the API Reference for complete function signatures.
- Report bugs and suggestions at https://github.com/chrplr/goxpyriment/issues.
- Happy experimenting!