```
1
2 static int A0mu[2076] = {
3
4   0 , 5 , 10 , 15 , 20 , 25 , 30 , 35 , 40 , 45 , 50 , 55 , 60 , 62 , 65 , 67 , 69 , 71 ,
    74 , 76 , 78 , 80 , 83 , 85 , 87 , 90 , 92 , 94 , 96 , 99 , 101 , 103 , 105 , 108 , 110
   , 112 , 113 , 115 , 116 , 118 , 119 , 121 , 122 , 124 , 125 , 127 , 128 , 130 , 132 , 133
    , 135 , 136 , 138 , 139 , 141 , 142 , 144 , 145 , 147 , 148 , 150 , 151 , 153 , 154 ,
   156 , 157 , 159 , 160 , 164 , 168 , 171 , 175 , 179 , 183 , 186 , 190 , 192 , 193 , 195 ,
    197 , 199 , 200 , 202 , 204 , 206 , 207 , 209 , 211 , 213 , 214 , 216 , 218 , 220 , 221
   , 223 , 225 , 227 ,
5   228 , 230 , 232 , 234 , 236 , 238 , 240 , 242 , 244 , 246 , 248 , 250 , 252 , 254 , 255
    , 257 , 259 , 261 , 263 , 265 , 267 , 269 , 271 , 273 , 275 , 277 , 279 , 281 , 283 ,
   285 , 287 , 289 , 291 , 293 , 295 , 297 , 299 , 301 , 303 , 305 , 306 , 308 , 310 , 312 ,
    314 , 316 , 318 , 320 , 322 , 324 , 326 , 328 , 330 , 333 , 335 , 338 , 340 , 343 , 345
   , 348 , 350 , 353 , 355 , 358 , 360 , 363 , 365 , 368 , 371 , 373 , 376 , 379 , 382 , 384
    , 387 , 390 , 392 , 395 , 398 , 400 , 403 , 406 , 408 , 411 , 414 , 417 , 419 , 422 ,
   425 , 427 , 430 , 433 ,
6   435 , 438 , 441 , 444 , 446 , 449 , 452 , 454 , 457 , 460 , 462 , 465 , 468 , 471 , 473
    , 476 , 479 , 481 , 484 , 487 , 489 , 492 , 495 , 498 , 500 , 503 , 506 , 508 , 511 ,
   514 , 516 , 519 , 522 , 525 , 527 , 530 , 531 , 533 , 534 , 535 , 537 , 538 , 540 , 541 ,
    542 , 544 , 545 , 546 , 548 , 549 , 550 , 552 , 553 , 555 , 556 , 557 , 559 , 560 , 563
   , 565 , 568 , 570 , 573 , 576 , 578 , 581 , 583 , 586 , 589 , 591 , 594 , 596 , 599 , 601
    , 604 , 607 , 609 , 612 , 614 , 617 , 620 , 622 , 625 , 627 , 630 , 632 , 633 , 635 ,
   636 , 638 , 640 , 641 ,
7   643 , 644 , 646 , 648 , 649 , 651 , 652 , 654 , 656 , 657 , 659 , 660 , 662 , 664 , 665
    , 667 , 668 , 670 , 673 , 676 , 679 , 682 , 685 , 688 , 691 , 694 , 697 , 700 , 703 ,
   706 , 709 , 712 , 715 , 718 , 721 , 724 , 727 , 730 , 732 , 734 , 736 , 738 , 740 , 742 ,
    744 , 746 , 748 , 750 , 752 , 754 , 756 , 758 , 760 , 763 , 766 , 769 , 771 , 774 , 777
   , 780 , 783 , 786 , 789 , 791 , 794 , 797 , 800 , 801 , 803 , 804 , 805 , 807 , 808 , 810
    , 811 , 812 , 814 , 815 , 816 , 818 , 819 , 820 , 822 , 823 , 825 , 826 , 827 , 829 ,
   830 , 833 , 835 , 838 ,
8   841 , 843 , 846 , 848 , 851 , 854 , 856 , 859 , 862 , 864 , 867 , 869 , 872 , 875 , 877
    , 880 , 883 , 885 , 888 , 890 , 893 , 895 , 898 , 900 , 903 , 905 , 908 , 910 , 913 ,
   915 , 918 , 920 , 923 , 925 , 928 , 930 , 932 , 934 , 936 , 939 , 941 , 943 , 945 , 947 ,
    949 , 951 , 954 , 956 , 958 , 960 , 962 , 963 , 965 , 966 , 968 , 969 , 971 , 972 , 974
   , 975 , 977 , 978 , 980 , 984 , 988 , 993 , 997 , 1001 , 1005 , 1009 , 1013 , 1018 , 1022
    , 1026 , 1030 , 1031 , 1033 , 1034 , 1035 , 1036 , 1038 , 1039 , 1040 , 1041 , 1043 ,
   1044 , 1045 , 1046 , 1048 ,
9   1049 , 1050 , 1052 , 1055 , 1057 , 1059 , 1062 , 1064 , 1066 , 1069 , 1071 , 1074 ,
   1076 , 1078 , 1081 , 1083 , 1085 , 1088 , 1090 , 1093 , 1095 , 1098 , 1100 , 1103 , 1105
   , 1108 , 1110 , 1113 , 1115 , 1118 , 1120 , 1122 , 1124 , 1126 , 1128 , 1131 , 1133 ,
   1135 , 1137 , 1139 , 1141 , 1143 , 1145 , 1147 , 1149 , 1152 , 1154 , 1156 , 1158 , 1160
   , 1162 , 1164 , 1166 , 1168 , 1170 , 1172 , 1174 , 1176 , 1178 , 1180 , 1182 , 1184 ,
   1187 , 1189 , 1191 , 1193 , 1196 , 1198 , 1200 , 1202 , 1204 , 1207 , 1209 , 1211 , 1213
   , 1216 , 1218 , 1220 , 1222 , 1224 ,
10  1227 , 1229 , 1231 , 1233 , 1236 , 1238 , 1240 , 1242 , 1244 , 1246 , 1248 , 1250 ,
   1252 , 1254 , 1256 , 1258 , 1260 , 1262 , 1264 , 1266 , 1268 , 1270 , 1272 , 1275 , 1277
   , 1279 , 1282 , 1284 , 1286 , 1289 , 1291 , 1294 , 1296 , 1298 , 1301 , 1303 , 1305 ,
   1308 , 1310 , 1313 , 1315 , 1318 , 1320 , 1323 , 1325 , 1328 , 1330 , 1333 , 1337 , 1340
   , 1343 , 1347 , 1350 , 1353 , 1357 , 1360 , 1363 , 1365 , 1368 , 1371 , 1373 , 1376 ,
   1379 , 1381 , 1384 , 1387 , 1389 , 1392 , 1395 , 1397 , 1400 , 1402 , 1404 , 1406 , 1408
   , 1409 , 1411 , 1413 , 1415 , 1417 ,
11  1419 , 1421 , 1423 , 1424 , 1426 , 1428 , 1430 , 1433 , 1435 , 1438 , 1440 , 1443 ,
   1445 , 1448 , 1450 , 1453 , 1455 , 1458 , 1460 , 1463 , 1467 , 1470 , 1473 , 1477 , 1480
   , 1483 , 1487 , 1490 , 1492 , 1494 , 1495 , 1497 , 1499 , 1501 , 1503 , 1505 , 1506 ,
   1508 , 1510 , 1512 , 1514 , 1515 , 1517 , 1519 , 1521 , 1523 , 1525 , 1526 , 1528 , 1530
   , 1533 , 1537 , 1540 , 1543 , 1547 , 1550 , 1553 , 1557 , 1560 , 1564 , 1567 , 1571 ,
   1575 , 1578 , 1582 , 1585 , 1589 , 1593 , 1596 , 1600 , 1602 , 1604 , 1606 , 1608 , 1609
   , 1611 , 1613 , 1615 , 1617 , 1619 ,
12  1621 , 1623 , 1624 , 1626 , 1628 , 1630 , 1633 , 1636 , 1639 , 1642 , 1645 , 1648 ,
   1651 , 1654 , 1657 , 1660 , 1663 , 1666 , 1669 , 1672 , 1675 , 1678 , 1681 , 1684 , 1687
   , 1690 , 1693 , 1697 , 1700 , 1703 , 1707 , 1710 , 1712 , 1713 , 1715 , 1717 , 1718 ,
   1720 , 1722 , 1723 , 1725 , 1727 , 1728 , 1730 , 1733 , 1735 , 1738 , 1741 , 1744 , 1746
   , 1749 , 1752 , 1755 , 1757 , 1760 , 1763 , 1765 , 1768 , 1770 , 1773 , 1775 , 1778 ,
   1780 , 1783 , 1785 , 1788 , 1790 , 1792 , 1794 , 1797 , 1799 , 1801 , 1803 , 1806 , 1808
   , 1810 , 1812 , 1814 , 1817 , 1819 ,
13  1821 , 1823 , 1826 , 1828 , 1830 , 1832 , 1835 , 1837 , 1839 , 1841 , 1844 , 1846 ,
```

```
13 1848 , 1850 , 1853 , 1855 , 1857 , 1860 , 1862 , 1864 , 1866 , 1869 , 1871 , 1873 , 1875
   , 1878 , 1880 , 1883 , 1886 , 1889 , 1892 , 1895 , 1898 , 1901 , 1904 , 1906 , 1909 ,
   1912 , 1915 , 1918 , 1921 , 1924 , 1927 , 1930 , 1934 , 1937 , 1941 , 1945 , 1948 , 1952
   , 1955 , 1959 , 1963 , 1966 , 1970 , 1973 , 1975 , 1978 , 1980 , 1983 , 1986 , 1988 ,
   1991 , 1993 , 1996 , 1999 , 2001 , 2004 , 2007 , 2009 , 2012 , 2014 , 2017 , 2020 , 2022
   , 2025 , 2027 , 2030 , 2033 , 2036 ,
14   2039 , 2042 , 2045 , 2048 , 2051 , 2054 , 2056 , 2059 , 2062 , 2065 , 2068 , 2071 ,
   2074 , 2077 , 2080 , 2082 , 2085 , 2087 , 2090 , 2092 , 2094 , 2097 , 2099 , 2101 , 2104
   , 2106 , 2109 , 2111 , 2113 , 2116 , 2118 , 2120 , 2123 , 2125 , 2128 , 2130 , 2133 ,
   2135 , 2138 , 2141 , 2144 , 2146 , 2149 , 2152 , 2155 , 2157 , 2160 , 2162 , 2165 , 2167
   , 2170 , 2172 , 2174 , 2177 , 2179 , 2182 , 2184 , 2187 , 2189 , 2191 , 2194 , 2196 ,
   2199 , 2201 , 2203 , 2206 , 2208 , 2211 , 2213 , 2216 , 2218 , 2220 , 2223 , 2225 , 2228
   , 2230 , 2233 , 2236 , 2239 , 2243 ,
15   2246 , 2249 , 2252 , 2255 , 2258 , 2261 , 2264 , 2268 , 2271 , 2274 , 2277 , 2280 ,
   2283 , 2286 , 2290 , 2293 , 2296 , 2299 , 2302 , 2306 , 2309 , 2312 , 2315 , 2318 , 2322
   , 2325 , 2328 , 2331 , 2334 , 2338 , 2341 , 2344 , 2347 , 2350 , 2354 , 2357 , 2360 ,
   2364 , 2368 , 2371 , 2375 , 2379 , 2383 , 2387 , 2390 , 2394 , 2398 , 2402 , 2406 , 2410
   , 2413 , 2417 , 2421 , 2425 , 2429 , 2432 , 2436 , 2440 , 2442 , 2443 , 2445 , 2446 ,
   2448 , 2450 , 2451 , 2453 , 2454 , 2456 , 2458 , 2459 , 2461 , 2462 , 2464 , 2466 , 2467
   , 2469 , 2470 , 2472 , 2474 , 2475 ,
16   2477 , 2478 , 2480 , 2484 , 2488 , 2493 , 2497 , 2501 , 2505 , 2509 , 2513 , 2518 ,
   2522 , 2526 , 2530 , 2533 , 2537 , 2540 , 2543 , 2547 , 2550 , 2553 , 2557 , 2560 , 2564
   , 2568 , 2571 , 2575 , 2579 , 2583 , 2586 , 2590 , 2593 , 2597 , 2600 , 2603 , 2607 ,
   2610 , 2613 , 2617 , 2620 , 2623 , 2627 , 2630 , 2633 , 2635 , 2638 , 2641 , 2644 , 2646
   , 2649 , 2652 , 2655 , 2657 , 2660 , 2663 , 2666 , 2669 , 2671 , 2674 , 2677 , 2680 ,
   2684 , 2688 , 2693 , 2697 , 2701 , 2705 , 2709 , 2713 , 2718 , 2722 , 2726 , 2730 , 2733
   , 2735 , 2738 , 2740 , 2743 , 2745 ,
17   2748 , 2750 , 2753 , 2755 , 2758 , 2760 , 2765 , 2770 , 2775 , 2780 , 2785 , 2790 ,
   2793 , 2796 , 2799 , 2802 , 2805 , 2808 , 2811 , 2814 , 2817 , 2820 , 2823 , 2827 , 2830
   , 2833 , 2837 , 2840 , 2843 , 2847 , 2850 , 2853 , 2857 , 2860 , 2863 , 2867 , 2870 ,
   2873 , 2877 , 2880 , 2883 , 2886 , 2889 , 2893 , 2896 , 2899 , 2902 , 2905 , 2908 , 2911
   , 2914 , 2918 , 2921 , 2924 , 2927 , 2930 , 2934 , 2939 , 2943 , 2948 , 2952 , 2957 ,
   2961 , 2966 , 2970 , 2973 , 2976 , 2979 , 2982 , 2985 , 2988 , 2991 , 2994 , 2996 , 2999
   , 3002 , 3005 , 3008 , 3011 , 3014 ,
18   3017 , 3020 , 3024 , 3028 , 3031 , 3035 , 3039 , 3043 , 3046 , 3050 , 3054 , 3058 ,
   3061 , 3065 , 3069 , 3073 , 3076 , 3080 , 3084 , 3087 , 3091 , 3094 , 3098 , 3101 , 3105
   , 3108 , 3112 , 3115 , 3119 , 3122 , 3126 , 3129 , 3133 , 3136 , 3140 , 3144 , 3147 ,
   3151 , 3155 , 3158 , 3162 , 3165 , 3169 , 3173 , 3176 , 3180 , 3183 , 3186 , 3189 , 3193
   , 3196 , 3199 , 3202 , 3205 , 3208 , 3211 , 3214 , 3218 , 3221 , 3224 , 3227 , 3230 ,
   3234 , 3238 , 3242 , 3246 , 3250 , 3254 , 3258 , 3262 , 3266 , 3270 , 3274 , 3278 , 3282
   , 3286 , 3290 , 3294 , 3297 , 3301 ,
19   3305 , 3308 , 3312 , 3315 , 3319 , 3323 , 3326 , 3330 , 3335 , 3340 , 3345 , 3350 ,
   3355 , 3360 , 3365 , 3370 , 3374 , 3378 , 3383 , 3387 , 3391 , 3395 , 3399 , 3403 , 3408
   , 3412 , 3416 , 3420 , 3425 , 3430 , 3435 , 3440 , 3445 , 3450 , 3455 , 3460 , 3463 ,
   3466 , 3469 , 3472 , 3475 , 3478 , 3481 , 3484 , 3487 , 3490 , 3497 , 3503 , 3510 , 3512
   , 3514 , 3517 , 3519 , 3521 , 3523 , 3526 , 3528 , 3530 , 3536 , 3541 , 3547 , 3552 ,
   3558 , 3563 , 3569 , 3574 , 3580 , 3584 , 3588 , 3591 , 3595 , 3599 , 3603 , 3606 , 3610
   , 3614 , 3617 , 3621 , 3625 , 3628 ,
20   3632 , 3635 , 3639 , 3643 , 3646 , 3650 , 3654 , 3657 , 3661 , 3665 , 3668 , 3672 ,
   3675 , 3679 , 3683 , 3686 , 3690 , 3696 , 3701 , 3707 , 3713 , 3719 , 3724 , 3730 , 3734
   , 3738 , 3743 , 3747 , 3751 , 3755 , 3759 , 3763 , 3768 , 3772 , 3776 , 3780 , 3786 ,
   3792 , 3798 , 3804 , 3810 , 3813 , 3816 , 3819 , 3821 , 3824 , 3827 , 3830 , 3836 , 3843
   , 3849 , 3855 , 3861 , 3868 , 3874 , 3880 , 3884 , 3888 , 3891 , 3895 , 3899 , 3903 ,
   3906 , 3910 , 3915 , 3920 , 3925 , 3930 , 3935 , 3940 , 3945 , 3950 , 3956 , 3963 , 3969
   , 3975 , 3981 , 3988 , 3994 , 4000 ,
21   4002 , 4005 , 4007 , 4009 , 4012 , 4014 , 4016 , 4018 , 4021 , 4023 , 4025 , 4028 ,
   4030 , 4037 , 4044 , 4051 , 4059 , 4066 , 4073 , 4080 , 4084 , 4089 , 4093 , 4098 , 4102
   , 4107 , 4111 , 4116 , 4120 , 4127 , 4134 , 4141 , 4148 , 4155 , 4162 , 4169 , 4176 ,
   4183 , 4190 , 4194 , 4198 , 4202 , 4206 , 4210 , 4214 , 4218 , 4222 , 4226 , 4230 , 4234
   , 4238 , 4242 , 4246 , 4250 , 4260 , 4270 , 4280 , 4290 , 4295 , 4299 , 4304 , 4308 ,
   4313 , 4317 , 4322 , 4326 , 4331 , 4335 , 4340 , 4345 , 4351 , 4356 , 4361 , 4366 , 4372
   , 4377 , 4382 , 4388 , 4393 , 4398 ,
22   4404 , 4409 , 4414 , 4419 , 4425 , 4430 , 4438 , 4445 , 4453 , 4460 , 4467 , 4473 ,
   4480 , 4487 , 4493 , 4500 , 4503 , 4507 , 4510 , 4520 , 4530 , 4540 , 4550 , 4560 , 4570
   , 4580 , 4584 , 4588 , 4592 , 4595 , 4599 , 4603 , 4607 , 4611 , 4615 , 4618 , 4622 ,
   4626 , 4630 , 4636 , 4641 , 4647 , 4653 , 4659 , 4664 , 4670 , 4676 , 4681 , 4687 , 4693
   , 4699 , 4704 , 4710 , 4716 , 4722 , 4728 , 4734 , 4740 , 4746 , 4752 , 4758 , 4764 ,
```

```
22 4770 , 4777 , 4784 , 4791 , 4798 , 4805 , 4812 , 4819 , 4826 , 4833 , 4840 , 4844 , 4849
   , 4853 , 4858 , 4862 , 4867 , 4871 ,
23   4876 , 4880 , 4888 , 4895 , 4903 , 4910 , 4918 , 4925 , 4933 , 4940 , 4943 , 4947 ,
   4950 , 4953 , 4957 , 4960 , 4964 , 4968 , 4972 , 4976 , 4979 , 4983 , 4987 , 4991 , 4995
   , 4999 , 5003 , 5007 , 5011 , 5014 , 5018 , 5022 , 5026 , 5030 , 5050 , 5070 , 5090 ,
   5110 , 5130 , 5135 , 5141 , 5146 , 5152 , 5157 , 5162 , 5168 , 5173 , 5178 , 5184 , 5189
   , 5195 , 5200 , 5209 , 5218 , 5227 , 5236 , 5244 , 5253 , 5262 , 5271 , 5280 , 5285 ,
   5290 , 5295 , 5300 , 5305 , 5310 , 5315 , 5320 , 5325 , 5330 , 5335 , 5340 , 5348 , 5356
   , 5365 , 5373 , 5381 , 5389 , 5397 ,
24   5405 , 5414 , 5422 , 5430 , 5437 , 5444 , 5451 , 5459 , 5466 , 5473 , 5480 , 5487 ,
   5494 , 5501 , 5509 , 5516 , 5523 , 5530 , 5538 , 5547 , 5555 , 5563 , 5572 , 5580 , 5587
   , 5594 , 5601 , 5608 , 5615 , 5622 , 5629 , 5636 , 5644 , 5651 , 5658 , 5665 , 5672 ,
   5679 , 5686 , 5693 , 5700 , 5711 , 5723 , 5734 , 5746 , 5757 , 5769 , 5780 , 5787 , 5795
   , 5802 , 5809 , 5816 , 5824 , 5831 , 5838 , 5845 , 5853 , 5860 , 5869 , 5878 , 5887 ,
   5896 , 5904 , 5913 , 5922 , 5931 , 5940 , 5945 , 5950 , 5955 , 5960 , 5965 , 5969 , 5974
   , 5979 , 5983 , 5988 , 5993 , 5997 ,
25   6002 , 6007 , 6011 , 6016 , 6021 , 6025 , 6030 , 6046 , 6063 , 6079 , 6095 , 6111 ,
   6128 , 6144 , 6160 , 6169 , 6178 , 6186 , 6195 , 6204 , 6213 , 6221 , 6230 , 6239 , 6248
   , 6256 , 6265 , 6274 , 6283 , 6291 , 6300 , 6313 , 6326 , 6339 , 6351 , 6364 , 6377 ,
   6390 , 6399 , 6407 , 6416 , 6425 , 6433 , 6442 , 6451 , 6459 , 6468 , 6477 , 6485 , 6494
   , 6503 , 6511 , 6520 , 6529 , 6538 , 6547 , 6556 , 6564 , 6573 , 6582 , 6591 , 6600 ,
   6610 , 6620 , 6630 , 6640 , 6650 , 6660 , 6670 , 6680 , 6690 , 6700 , 6710 , 6720 , 6730
   , 6740 , 6750 , 6764 , 6779 , 6793 ,
26   6807 , 6821 , 6836 , 6850 , 6862 , 6874 , 6887 , 6899 , 6911 , 6923 , 6936 , 6948 ,
   6960 , 6968 , 6975 , 6983 , 6990 , 6998 , 7005 , 7013 , 7020 , 7028 , 7035 , 7043 , 7050
   , 7062 , 7074 , 7086 , 7098 , 7109 , 7121 , 7133 , 7145 , 7157 , 7169 , 7181 , 7193 ,
   7204 , 7216 , 7228 , 7240 , 7254 , 7268 , 7282 , 7296 , 7310 , 7324 , 7338 , 7352 , 7366
   , 7380 , 7397 , 7413 , 7430 , 7447 , 7463 , 7480 , 7493 , 7505 , 7518 , 7530 , 7543 ,
   7555 , 7568 , 7580 , 7593 , 7605 , 7618 , 7630 , 7643 , 7655 , 7668 , 7680 , 7695 , 7711
   , 7726 , 7742 , 7757 , 7772 , 7788 ,
27   7803 , 7818 , 7834 , 7849 , 7865 , 7880 , 7894 , 7907 , 7921 , 7935 , 7948 , 7962 ,
   7975 , 7989 , 8003 , 8016 , 8030 , 8047 , 8064 , 8081 , 8098 , 8115 , 8132 , 8149 , 8166
   , 8183 , 8200 , 8214 , 8228 , 8241 , 8255 , 8269 , 8283 , 8296 , 8310 , 8326 , 8343 ,
   8359 , 8375 , 8391 , 8408 , 8424 , 8440 , 8456 , 8473 , 8489 , 8505 , 8521 , 8538 , 8554
   , 8570 , 8590 , 8610 , 8630 , 8650 , 8670 , 8690 , 8710 , 8730 , 8750 , 8770 , 8791 ,
   8812 , 8833 , 8854 , 8876 , 8897 , 8918 , 8939 , 8960 , 8970 , 8980 , 8990 , 9000 , 9010
   , 9020 , 9030 , 9065 , 9100 , 9135 ,
28   9170 , 9193 , 9217 , 9240 , 9263 , 9287 , 9310 , 9333 , 9357 , 9380 , 9401 , 9422 ,
   9443 , 9464 , 9486 , 9507 , 9528 , 9549 , 9570 , 9592 , 9615 , 9637 , 9659 , 9682 , 9704
   , 9726 , 9748 , 9771 , 9793 , 9815 , 9838 , 9860 , 9886 , 9912 , 9937 , 9963 , 9989 ,
   10015 , 10041 , 10066 , 10092 , 10118 , 10144 , 10169 , 10195 , 10221 , 10247 , 10273 ,
   10298 , 10324 , 10350 , 10376 , 10403 , 10429 , 10455 , 10482 , 10508 , 10535 , 10561 ,
   10587 , 10614 , 10640 , 10677 , 10713 , 10750 , 10787 , 10823 , 10860 , 10891 , 10921 ,
   10952 , 10983 , 11013 , 11044 , 11075 ,
29   11105 , 11136 , 11167 , 11197 , 11228 , 11259 , 11289 , 11320 , 11359 , 11398 , 11437 ,
    11476 , 11515 , 11554 , 11593 , 11632 , 11671 , 11710 , 11960
30
31   };
32
33 #define range 480
34 #define sample_num 120
35
36 static int waveformsTable[sample_num] = {
37   // Sin wave
38
39     0x7ff, 0x86a, 0x8d5, 0x93f, 0x9a9, 0xa11, 0xa78, 0xadd, 0xb40, 0xba1,
40     0xbff, 0xc5a, 0xcb2, 0xd08, 0xd59, 0xda7, 0xdf1, 0xe36, 0xe77, 0xeb4,
41     0xeec, 0xf1f, 0xf4d, 0xf77, 0xf9a, 0xfb9, 0xfd2, 0xfe5, 0xff3, 0xffc,
42     0xfff, 0xffc, 0xff3, 0xfe5, 0xfd2, 0xfb9, 0xf9a, 0xf77, 0xf4d, 0xf1f,
43     0xeec, 0xeb4, 0xe77, 0xe36, 0xdf1, 0xda7, 0xd59, 0xd08, 0xcb2, 0xc5a,
44     0xbff, 0xba1, 0xb40, 0xadd, 0xa78, 0xa11, 0x9a9, 0x93f, 0x8d5, 0x86a,
45     0x7ff, 0x794, 0x729, 0x6bf, 0x655, 0x5ed, 0x586, 0x521, 0x4be, 0x45d,
46     0x3ff, 0x3a4, 0x34c, 0x2f6, 0x2a5, 0x257, 0x20d, 0x1c8, 0x187, 0x14a,
47     0x112, 0xdf, 0xb1, 0x87, 0x64, 0x45, 0x2c, 0x19, 0xb, 0x2,
48     0x0, 0x2, 0xb, 0x19, 0x2c, 0x45, 0x64, 0x87, 0xb1, 0xdf,
49     0x112, 0x14a, 0x187, 0x1c8, 0x20d, 0x257, 0x2a5, 0x2f6, 0x34c, 0x3a4,
50     0x3ff, 0x45d, 0x4be, 0x521, 0x586, 0x5ed, 0x655, 0x6bf, 0x729, 0x794
```

```
 51   };
 52
 53
 54 byte b[3] = {0,0,0}, mode, val_in[4] = {0,0,0,0};
 55 int measure = 0, run_option = 0;
 56 int func = 0, pos = 0, centre, peak, trough, pos_0 = 1558, mu, mu_tol = 40, set_strain =
     2048;
 57 int step_count = 0, sign_change_count = 0, strain_closing_in = 0, A0_amp, dA0dt_amp;
 58 int mean_amp_step = 0, max_amp_step = 0, mean_amp = 0, mean_tried = 0,
    amp_step_deviation = 0;
 59 int min_amp_step = 0, attempt_count = 0, ppc = 0;
 60 int past_step_estimates[8], past_amp_steps[8], past_amps[8], past_attempts[8], past_pos[
    3];
 61 int amp = 64, old_peak_to_trough, old_amp_step = 1, amp_step = 4, old_amp = 64, fiddle =
     2;
 62 int f = 0, p = 0, delta_num = 0, li=0, corr, t_diff = 0, f_count = 0;
 63 int t = 0, last_t = 0, t_peak, t_trough, dt, t_dpeakdt, t_dtroughdt, dpeakdt, dtroughdt;
 64 int dmudt, darraydt[range], DC_func = 2047, NR = 0, korb = 0, simu_k = 0, simu_b = 0,
    Torv = 0;
 65 int centre_mode = 0, equilibrium_A0 = -1, used_zero_A0, simu_k_unit = 128, simu_b_unit =
     128;
 66 int peak_to_peak, upper_amplitude, lower_amplitude, centre_estimated = 0;
 67 int freq_check = 0, pos_rec, A0_period_estimate_mean = 0, A0_period_count = 0;
 68 unsigned long int t_i = 0, t_f = 0, val = 70000;//35000;//175000*4;
 69 int array[range], peaksnt[8][4], troughsnt[8][4], rec_times[32], A0_period_estimates[16]
    ;
 70 int phase_estimates[16], pest_num = -1, feed_num = -1, sym_check, pest_check, e_num = -1
    ;
 71 int cycle_counter = 0;
 72 int mu_one_back, mu_two_back;
 73
 74 void handle_pc_input();
 75 void spike_filter();
 76 void handle_const_strain_feedback();
 77 void adaptive_step_calculation_for_const_strain();
 78 void settle_amp();
 79 void send_3_byte_value(int value, byte third_byte);
 80 void send_out_of_bounds_values();
 81 void send_func();
 82 void send_mu();
 83 void send_pos();
 84 void map_centre_back_to_pos(int last_pos, int try_num, int twos);
 85
 86 void setup(){
 87   SerialUSB.begin(115200);
 88   analogWriteResolution(12);  // set the analog output resolution to 12 bit (4096 levels
    )
 89   analogReadResolution(12);   // set the analog input resolution to 12 bit (4096 levels)
 90   pinMode(DAC1, OUTPUT);
 91   pinMode(22, OUTPUT);
 92   pinMode(2,OUTPUT);     // port B pin 25
 93   analogWrite(2,255);    // sets up some other registers I haven't worked out yet
 94   REG_PIOB_PDR = 1<<25; // disable PIO, enable peripheral
 95   REG_PIOB_ABSR= 1<<25; // select peripheral B
 96   REG_TC0_WPMR=0x54494D00; // enable write to registers
 97   REG_TC0_CMR0=0b00000000000001001110001000000000; // set channel mode register (see
    datasheet)
 98   REG_TC0_RC0= val; // counter period
 99   // needs to be HALF of (1/freq) * (84 MHz/sample_num) for some reason
100   // 2 is smallest prescale factor, that's why
101   // clock speed is 84 MHz
102   REG_TC0_RA0=30000000;  // PWM value
103   REG_TC0_CCR0=0b101;    // start counter
104   REG_TC0_IER0=0b00010000; // enable interrupt on counter=rc
105   REG_TC0_IDR0=0b11101111; // disable other interrupts
106
107   NVIC_EnableIRQ(TC0_IRQn); // enable TC0 interrupts
```

```
108
109   REG_PMC_WPMR = 0x504D43;
110   bitSet(REG_PMC_PCER1,6);
111   REG_DACC_CR = 1;
112   REG_DACC_WPMR = 0x444143;
113   REG_DACC_ACR = 0b00000000000000000000000100001010;
114   REG_DACC_IER = 1;
115   REG_DACC_MR = 0b00001100000000010000100000000000;
116   bitSet(REG_DACC_CHER,1);
117 }
118
119 void loop(){
120
121 }
122
123 void TC0_Handler(){
124
125     long dummy=REG_TC0_SR0; // vital - reading this clears some flag
126                             // otherwise you get infinite interrupts
127
128     //bitSet(REG_PIOB_SODR,26);
129
130     pos = analogRead(measure); //reads the voltage at analog pin A0
131
132     if( (pos >=1558 && pos <= 3633)){
133       int num = (pos - pos_0);
134       //mu_two_back = mu_one_back;
135       //mu_one_back = mu;
136       mu = A0mu[num];
137       //send_mu();
138   }
139
140     if(1 == centre_mode){used_zero_A0 = equilibrium_A0;}
141     else{used_zero_A0 = centre;}
142
143     if(1 != NR && (0 == run_option || 1 == run_option) ){
144     func = ( ( (waveformsTable[t] - waveformsTable[0]) * amp) / 2048 ) + 2047;
145     }
146     else if(1 != NR && 2 == run_option){func = DC_func;}
147     else if(1 == NR && (0 == run_option || 1 == run_option) ){
148     func = ( ( ( (waveformsTable[t] - waveformsTable[0]) * amp) / 2048 )
149          + ( (
150          //( (mu_two_back - used_zero_A0) + (mu_one_back - used_zero_A0) + (mu -
    used_zero_A0) )
151          //( (mu_one_back - used_zero_A0) + (mu - used_zero_A0) )
152          (mu - used_zero_A0)
153          * simu_k ) / (simu_k_unit)
154          )
155          + ( ( dmudt * simu_b ) / (simu_b_unit) ) )
156          + 2047 ;
157        }
158     else if(1 == NR && 2 == run_option){
159     func = DC_func + ( (
160          //( (mu_two_back - used_zero_A0) + (mu_one_back - used_zero_A0) + (mu -
    used_zero_A0) )
161          //( (mu_one_back - used_zero_A0) + (mu - used_zero_A0) )
162          (mu - used_zero_A0)
163          * simu_k ) / (simu_k_unit)
164          )
165            + ( ( dmudt * simu_b ) / (simu_b_unit) ) ;
166            }
167
168     //analogWrite(DAC1, func); //writes this sine wave to the DAC output pin 1
169     if(4095 < func){func = 4095;}
170     if(0 > func){func = 0;}
171     REG_DACC_CDR = func;
172
```

```
173     if(2 == NR && (0 == run_option || 1 == run_option)){
174     func = ( ( ( (waveformsTable[t] - waveformsTable[0]) * amp) / 2048 )
175          + ( (
176          ( (mu_one_back - used_zero_A0) + (mu - used_zero_A0) )
177          * simu_k ) / (simu_k_unit * 2)
178          )
179          + ( ( dmudt * simu_b ) / (simu_b_unit) ) )
180          + 2047 ;
181        }
182     else if(2 == NR && 2 == run_option){
183     func = DC_func + ( (
184          ( (mu_one_back - used_zero_A0) + (mu - used_zero_A0) )
185          * simu_k ) / (simu_k_unit * 2)
186          )
187                    + ( ( dmudt * simu_b ) / (simu_b_unit) ) ;
188            }//prioritise NR view over velocity view
189     else if(1 == Torv && 1 != NR){
190     func = dmudt + 2047;
191        }
192
193     send_func(); //send the DAC1 output value via SerialUSB
194     if( (pos >=1558 && pos <= 3633)){send_mu();}
195
196     //cycle_counter = 1 - cycle_counter;
197     //digitalWrite(42,cycle_counter);
198
199     handle_const_strain_feedback();
200
201     handle_pc_input();
202
203     t++; // counts number of completed computing cycles
204     if(t == sample_num){t = 0;}  // Reset the counter to repeat the wave
205
206     //bitSet(REG_PIOB_CODR,26);
207 }
208
209
210
211 // ~~~~~~~~~~~~ code outsourced into inelegant functions ~~~~~~~~~
212
213
214 // ~~~~~~~~~~ beginning of space for feedback functions ~~~~~~~~~~~~~
215
216 void handle_const_strain_feedback(){
217
218      if( (pos <= 1558 || pos >= 3633) && amp >= 1){
219        if(0 == run_option){amp -= 1;}
220        send_out_of_bounds_values();
221        if( 1 == freq_check ){
222         t_diff ++; // also count time of out of bounds events
223        }
224     }
225
226     else{
227
228       if( 1 == freq_check ){
229         if( (pos + 16) >= pos_rec && (pos - 16) <= pos_rec){//points are sparse,
230           // so a strict recurrence condition can miss many cycles
231           if( 175000 <= val ){//at freq < 2Hz shorter periods are presumably the 2Hz
    fundamental
232            if( 6 < t_diff ){
233             if( 0 == (f_count % 2) && 1 < f_count
234              && 15 > ( t_diff + rec_times[f_count] ) ){f_count -=2;}
235             else if( 0 == (f_count % 2) ){//2nd recurrence of a value implies completed
    period
236               A0_period_estimates[A0_period_count] = t_diff + rec_times[f_count];
237               pest_check = (120 - A0_period_estimates[A0_period_count]);
```

```
238               pest_num = A0_period_count;
239               A0_period_count ++;
240             }
241
242           rec_times[f_count] = t_diff;
243           f_count ++;
244           }
245           t_diff = 0;
246           }
247           else if( 175000 > val ){//at freq > 2Hz reject periods shorter than half a
    driving cycle
248             if( 6 < t_diff ){
249              if( 0 == (f_count % 2) && 1 <= f_count
250               && 60 > ( t_diff + rec_times[f_count] ) ){f_count -=2;}
251             else if( 0 == (f_count % 2) ){//2nd recurrence of a value implies completed
    period
252                A0_period_estimates[A0_period_count] = t_diff + rec_times[f_count];
253                pest_check = (120 - A0_period_estimates[A0_period_count]);
254                pest_num = A0_period_count;
255                A0_period_count ++;
256              }
257
258             rec_times[f_count] = t_diff;
259             f_count ++;
260             }
261             t_diff = 0;
262             }
263
264
265               if( 32 == f_count){
266                int two_rec_mean = 0;
267                 for(int i = 0; i < f_count; i++){
268                    two_rec_mean += rec_times[i];
269                  }
270                 A0_period_estimate_mean = (two_rec_mean/16);
271                 pest_check = (120 - A0_period_estimates[A0_period_count]);
272                 f_count = 0;
273                 freq_check = 2;
274               }
275               if( 16 == A0_period_count ){A0_period_count = 0;}
276          }
277        else if( 1600 < t_diff ){//handle time overflow for non-recurring pos values
278         freq_check = 2;//possibly guessed range doesn't contain value, so take centre
    instead
279          t_diff = 0;
280        }
281        else{t_diff ++;}
282      }/*
283      else if( (pos >= 2200 && pos <= 2900) && 0 == freq_check ){
284        freq_check = 1;
285        pos_rec = pos;
286        }*/
287
288    if(1 == centre_estimated ){
289
290    switch(p){
291    case 0 :send_3_byte_value(amp, 0b00000000);break;
292    case 3 :send_3_byte_value(sym_check, 0b00000001);break;
293    case 5 :send_3_byte_value(centre, 0b00000010);break;
294    case 7 :send_3_byte_value(peak_to_peak, 0b00000100);break;
295    case 9 :send_3_byte_value(A0_period_estimates[A0_period_count], 0b00001000);break;
296    case 11 :if( 0 > phase_estimates[delta_num] ){
297              send_3_byte_value(phase_estimates[delta_num], 0b00010001);
298            }
299              else {send_3_byte_value(phase_estimates[delta_num], 0b00010000);}
300              //centre_estimated = 0;
301              break;
```

```
302     case 125 :send_3_byte_value(amp, 0b00000000);break;
303     case 129 :send_3_byte_value(A0_period_estimates[A0_period_count], 0b00001000);break;
304     case 131 :if( 0 > phase_estimates[delta_num] ){
305               send_3_byte_value(phase_estimates[delta_num], 0b00010001);
306             }
307               else {send_3_byte_value(phase_estimates[delta_num], 0b00010000);}
308               break;
309     case 245 :send_3_byte_value(amp, 0b00000000);break;
310     case 249 :send_3_byte_value(A0_period_estimates[A0_period_count], 0b00001000);break;
311     case 251 :if( 0 > phase_estimates[delta_num] ){
312               send_3_byte_value(phase_estimates[delta_num], 0b00010001);
313             }
314               else {send_3_byte_value(phase_estimates[delta_num], 0b00010000);}
315               break;
316     case 365 :send_3_byte_value(amp, 0b00000000);break;
317     case 369 :send_3_byte_value(A0_period_estimates[A0_period_count], 0b00001000);break;
318     case 371 :if( 0 > phase_estimates[delta_num] ){
319               send_3_byte_value(phase_estimates[delta_num], 0b00010001);
320             }
321               else {send_3_byte_value(phase_estimates[delta_num], 0b00010000);}
322               centre_estimated = 0;
323               break;
324     }
325     //send_3_byte_values();
326
327     }
328      //mu = A0mu[pos - pos_0];
329
330     if(p < range){
331     array[p] = mu;
332
333     if(last_t > t){dt = t + (120-last_t);}
334        else {dt = t -last_t;}
335
336        if(0 < p && range > p){// time differential quotient (new - old) / dt
337          dmudt = ((array[p] - array[p-1])/dt);
338        }
339        else if(0 == p){
340          dmudt = ((array[0] - array[range-1])/dt);
341        }/*
342        else if(1 == p){
343          dmudt = ((array[p-1] - array[range-1])/dt);
344        }*/
345
346      darraydt[p] = dmudt;
347
348       if( (p % sample_num) == 1){// reset estimates every full cycle, for p == 1, 121,
    241 and 361
349
350          if( 0 <= e_num ){
351
352          peaksnt[e_num][0] = peak;
353          peaksnt[e_num][1] = t_peak;
354          peaksnt[e_num][2] = dpeakdt;
355          peaksnt[e_num][3] = t_dpeakdt;
356          troughsnt[e_num][0] = trough;
357          troughsnt[e_num][1] = t_trough;
358          troughsnt[e_num][2] = dtroughdt;
359          troughsnt[e_num][3] = t_dtroughdt;
360
361          A0_amp = ( (peaksnt[feed_num][0] - troughsnt[feed_num][0]) /2);
362          dA0dt_amp = ( (peaksnt[feed_num][2] - troughsnt[feed_num][2]) /2);
363          // positive phase_estimate means that the response leads the driving
364
365          if( 0 == (delta_num % 2) ){ // even numbered elements are estimates from peaks
366          phase_estimates[delta_num] = (30 - t_peak);
367          if( -60 > phase_estimates[delta_num] ){phase_estimates[delta_num] = sample_num
```

```
367 + phase_estimates[delta_num];}
368          else if( 60 < phase_estimates[delta_num] ){phase_estimates[delta_num] =
    sample_num - phase_estimates[delta_num];}
369          delta_num ++;
370          }
371
372          else{ // odd numbered elements are estimates from troughs
373          phase_estimates[delta_num] = (90 - t_trough);
374          if( -60 > phase_estimates[delta_num] ){phase_estimates[delta_num] = sample_num
    + phase_estimates[delta_num];}
375          else if( 60 < phase_estimates[delta_num] ){phase_estimates[delta_num] =
    sample_num - phase_estimates[delta_num];}
376          delta_num ++;
377          }
378
379        if (16 == delta_num){delta_num = 0;}
380          feed_num = e_num;
381          }
382          e_num ++;
383          if (8 == e_num){e_num = 0;}
384
385
386            if( array[p] > array[p-1]){ //start making new extremum estimates
387                 peak = array[p]; trough = array[p-1];
388                 t_peak = t;      t_trough = last_t;}
389            if( array[p] < array[p-1]){
390                 peak = array[p-1]; trough = array[p];
391                 t_peak = last_t;      t_trough = t;}
392
393            if( darraydt[p] > darraydt[p-1]){
394                 dpeakdt = darraydt[p]; dtroughdt = darraydt[p-1];
395                 t_dpeakdt = t;      t_dtroughdt = last_t;}
396            if( darraydt[p] < darraydt[p-1]){
397                 dpeakdt = array[p-1]; dtroughdt = darraydt[p];
398                 t_dpeakdt = last_t;      t_dtroughdt = t;}
399
400
401        }
402        else if( 1 < p && (p % sample_num) != 1){ //else compare values within cycle
403            if( peak < array[p]){peak = array[p]; t_peak = t;}
404            if( trough > array[p]){trough = array[p]; t_trough = t;}
405            if( dpeakdt < darraydt[p]){dpeakdt = darraydt[p]; t_dpeakdt = t;}
406            if( dtroughdt > darraydt[p]){dtroughdt = darraydt[p]; t_dtroughdt = t;}
407
408        }
409
410     p++;
411     }
412
413
414
415
416     if(p == range){// once sampled over 4 cycles, extract information
417       int mean = 0;
418       for(int j = 0; j < p; j++){
419         mean += array[j];
420         }
421       centre = (mean/p);
422       centre_estimated = 1;
423
424
425
426       peak_to_peak = (peak - trough);
427       upper_amplitude = (peak - centre);
428       lower_amplitude = (centre - trough);
429
430
```

```
      sym_check = (upper_amplitude - lower_amplitude);

      //Serial.println(centre);
      //Serial.println(peak);
      p = 0;
      //old_peak_to_trough = peak - trough;


      //if( 2 == freq_check || 1 == freq_check){
        map_centre_back_to_pos(1038, 0, 1);
        freq_check = 1;
        //}

      }


      last_t = t;
     }
    if(0 == run_option){


             if( 0x7ff == waveformsTable[t] && 0 <= feed_num ){


               // assess whether peak and trough values lie symmetrically about the centre value
               // if not, response likely hasn't settled to a steady state, so just wait
                 if( ( (4 * mu_tol) >= sym_check && ( (-4) * mu_tol) <= sym_check )
                    || ( 0 >= pest_num && 40 >= pest_check && (-40) <= pest_check ) ){
           //compare using last 1 cycle estimates
           if(0 == strain_closing_in ){adaptive_step_calculation_for_const_strain();}
           else if(1 == strain_closing_in ){settle_amp();}


           // increase amplitude, when it is lower than required
           if( ( ( (peaksnt[feed_num][0] - troughsnt[feed_num][0]) /2) <= set_strain - mu_tol ) && 2048 >= (amp + amp_step) ){
              amp += amp_step;
              old_amp_step = amp_step;
             }


           // decrease amplitude, when it is higher than required
           else if( ( ( (peaksnt[feed_num][0] - troughsnt[feed_num][0]) /2) >= set_strain + mu_tol ) && 0 <= (amp - amp_step) ){
              amp -= amp_step;
              old_amp_step = amp_step;
             }
          }
       }

    }

  }

void adaptive_step_calculation_for_const_strain(){

//( (waveformsTable[t] * 512 * amp) / 1048576 )

    if( 0 != old_amp_step && 1 <= feed_num ){
      fiddle = (128/old_amp_step);
      if( 64 < old_amp_step && 96 > old_amp_step){fiddle = 2;}
      else if( 96 < old_amp_step){fiddle = 1;}
      signed int step_estimate;

      step_estimate =( ( fiddle * old_amp_step * (
                      ( 2 * set_strain )
```

```
495                     - ( 2 * (peaksnt[feed_num][0] - troughsnt[feed_num][0]) )
496                     + ( peaksnt[feed_num - 1][0] - troughsnt[feed_num - 1][0]) ) )
497                     / (4*set_strain) );
498
499       past_step_estimates[step_count] = step_estimate;
500
501       if(8 >= amp && 8 < step_estimate){amp_step = 8;}
502
503       if(0 > step_estimate){
504        if(0 < step_count && 0 < past_step_estimates[step_count - 1]){sign_change_count
    ++;}
505
506       step_estimate = (-step_estimate);
507
508      }
509       else if(0 < step_count && 0 > past_step_estimates[step_count - 1]){
    sign_change_count ++;}
510
511       if( (2048 >= (amp + step_estimate) ) && (0 <= (amp - step_estimate) ) ){
512         amp_step = step_estimate;
513
514         }
515
516       else if( (2048 >= (amp + (step_estimate / 2) ) ) && (0 <= (amp - (step_estimate /
    2) ) ) ){
517         amp_step = (step_estimate / 2);
518         }
519
520       else if( (2048 >= (amp + (step_estimate / 4) ) ) && (0 <= (amp - (step_estimate /
    4) ) ) ){
521         amp_step = (step_estimate / 4);
522         }
523
524       else{ amp_step = 1; }
525
526
527       }
528
529     else if( 0 == old_amp_step && 1 >= feed_num ){
530
531       if( ( ( ( (peaksnt[feed_num][0] - troughsnt[feed_num][0]) /2) <= set_strain -
    mu_tol ) && 2048 > amp )
532        || ( ( ( (peaksnt[feed_num][0] - troughsnt[feed_num][0]) /2) >= set_strain +
    mu_tol ) && 0 < amp ) ){ amp_step = 1; }
533
534       }
535
536
537     if(0 < sign_change_count){
538     past_amp_steps[step_count] = amp_step;
539     step_count ++;
540     }
541
542     if(8 == step_count){
543       if(3 <= sign_change_count){
544         int d_amp_step = 0, d_amp = 0;
545         for(int i = 0; i < step_count;i++){
546           d_amp_step += past_amp_steps[i];
547           d_amp += past_amps[i];
548            if(1 <= i){
549             max_amp_step = max(past_amp_steps[i],past_amp_steps[i-1]);
550             min_amp_step = min(past_amp_steps[i],past_amp_steps[i-1]);
551             }
552           }
553        mean_amp = (d_amp/step_count);
554        mean_amp_step = (d_amp_step/step_count);
555       int d_step_dev = 0;
```

```
556       for(int i = 0; i < step_count;i++){
557         if(mean_amp_step >= past_amp_steps[i]){d_step_dev += (mean_amp_step -
    past_amp_steps[i]);}
558         else if(mean_amp_step < past_amp_steps[i]){d_step_dev += (past_amp_steps[i] -
    mean_amp_step);}
559       }
560        amp_step_deviation = (d_step_dev/step_count);
561        strain_closing_in = 1;
562         }
563     sign_change_count = 0;
564     step_count = 0;
565     }
566
567   }
568
569
570
571 void settle_amp(){
572  //int step_attempt = 0;
573  //int sign = 1;
574
575  if(0 == mean_tried){
576      amp = mean_amp;
577      mean_tried = 1;
578      }
579
580   if( set_strain < ( 2 * set_strain - (peaksnt[feed_num][0] - troughsnt[feed_num][0]) )
581    || (-1)*set_strain > ( 2 * set_strain - (peaksnt[feed_num][0] - troughsnt[feed_num][0
    ]) ) ){strain_closing_in = 0;}
582
583  amp_step = 1;
584  /*
585  if(3 >= sign_change_count){
586
587  if(  ( ( (peaksnt[feed_num][0] - troughsnt[feed_num][0]) /2) < set_strain ) ){sign = -1
    ;}
588
589
590   if(32 >= amp_step_deviation && 4 > attempt_count){step_attempt = (amp_step_deviation/4
    );}
591   else if(16 >= amp_step_deviation && 4 > attempt_count){step_attempt = 4;}
592   else{step_attempt = 2;}
593
594   if(0 < attempt_count){
595     if( (0 < past_attempts[attempt_count] && 0 > past_attempts[attempt_count - 1] )
596      || (0 > past_attempts[attempt_count] && 0 < past_attempts[attempt_count - 1] ) ){
    sign_change_count ++;}
597   }
598
599   if( (mean_amp + mean_amp_step) <= 2048 && (mean_amp + mean_amp_step) >= (amp +
    step_attempt)
600    && (mean_amp - mean_amp_step) <= (amp - step_attempt) && mean_amp > mean_amp_step
     ){
601    past_attempts[attempt_count] = sign*step_attempt;
602    amp_step = step_attempt;}
603
604   attempt_count ++;
605   }
606
607   else{amp_step = 1;}
608
609   if(8 == attempt_count){
610   sign_change_count = 0;
611   attempt_count = 0;
612    }
613    */
614   }
```

```
615
616
617 // ~~~~~~~~~~ end of space for feedback functions ~~~~~~~~~~~~~~
618
619
620
621 void handle_pc_input(){
622   if(SerialUSB.available() > 0){
623       mode = SerialUSB.read(); //first byte encodes input type
624         //REG_TC0_RC0 *= 2;
625        if(mode == 0 || mode == 1 || mode == 3 || mode == 5){
626         SerialUSB.readBytes(val_in,4);
627         for(int i = 0; i < 8; i++){
628           for(int j = 0; j < 4; j++){
629              bitWrite(val,i+j*8,bitRead(val_in[j],i));
630             }
631            }
632            //if(val == 1024){
633            //REG_TC0_RC0 *= 2;
634            //}
635            if(mode == 0){
636          REG_TC0_RC0 = val;
637          // cannot predict response at new frequency
638          // hence set to zero
639          if(0 == run_option){amp = 0;}
640          amp_step = 0;
641          old_amp_step = 0;
642          strain_closing_in = 0;
643          // interrupt handler intervals have changed size
644          // restart data aqcuisition processes
645          freq_check = 0;
646          p = 0;
647          A0_period_count = 0;
648          t_diff = 0;
649          e_num = -1;
650          delta_num = 0;
651          pest_num = -1;
652          feed_num = -1;
653          }
654         else if(mode == 1){//bitRead(mode,0) == 1)
655
656          if(0 == run_option){
657          set_strain = val;
658          amp_step = 0;
659          old_amp_step = 0;
660          strain_closing_in = 0;
661            }
662          else if(1 == run_option){
663          amp = val;
664            }
665          else if(2 == run_option){
666          DC_func = val;
667            }
668          }
669
670         else if(mode == 3){
671          if(0 == korb){
672          simu_k = val;
673            }
674          else if(1 == korb){
675          simu_b = val;
676            }
677          }
678         else if(mode == 5){
679          equilibrium_A0 = val;
680          }
681        }
```

```
682       else if( (mode == 0b00000010 || mode == 0b00000100 )
683             && (1 == run_option || 2 == run_option) ){
684        if(mode == 0b00000010){
685
686          if(1 == run_option && 0 < amp){amp -= 1;}
687          if(2 == run_option && DC_func > 0){DC_func -= 1;}
688          //amp_step /= 2;
689          }
690
691        else if(mode == 0b00000100){
692
693          if(1 == run_option && 2048 > amp){amp += 1;}
694          if(2 == run_option && DC_func < 4095){DC_func += 1;}
695         /* amp_step *= 2;
696         if( amp_step == 0){
697          amp_step = 1;
698         }*/
699          }
700        }
701         else if(mode == 0b00001000){
702          //attempt to keep strain amplitude constant
703          run_option = 0;
704          amp_step = 0;
705          old_amp_step = 0;
706          strain_closing_in = 0;
707          }
708
709         else if(mode == 0b00010000){
710          //keep stress amplitude constant
711          run_option = 1;
712          }
713
714         else if(mode == 0b00100000){
715          //deactivate spring and damping simulation
716          NR = 0;
717          }
718
719         else if(mode == 0b01000000){
720          //activate spring and damping simulation
721          NR = 1;
722          }
723
724          else if(mode == 0b00000111){
725          //send a DC current to oscillator
726          run_option = 2;
727          amp = 0;
728          }
729
730          else if(mode == 0b00000110){
731          //toggle spring/damping value user input
732          korb = 1 - korb;
733          }
734
735          else if(mode == 0b00001100){
736          //only send simulation via USB, but do not apply
737          NR = 2;
738          }
739
740          else if(mode == 0b00011000){
741          //toggle driving torgue/measured velocity being sent via USB
742          Torv = 1 - Torv;
743          }
744
745          else if(mode == 0b00110000){
746          //toggle floating centre/permanent equilibrium displacement estimate
747          centre_mode = 1 - centre_mode;
748          if(-1 == equilibrium_A0){equilibrium_A0 = centre;}
```

```
749          }
750
751
752        }
753
754    }
755
756 // ~~~~~~~~~~ beginning of space for functions that send data via SerialUSB ~~~~~~~~~~
757
758 void send_3_byte_value(int value, byte third_byte){
759    // give 3rd byte in binary 0b7th6th5th4th3rd2nd1st0th
760
761    b[0] = value;
762
763    for(int i = 8; i <= 14; i++){ //had problems with 14th bit being set for no reason
764     bitWrite(b[1],i-8,bitRead(value,i));
765     }
766
767     bitSet(b[1],7);
768     b[2] = third_byte;
769     SerialUSB.write(b,3);
770     bitClear(b[1],7);
771     bitClear(b[1],6);
772     b[2] = 0;
773
774    }
775
776
777  void send_out_of_bounds_values(){
778    int excess;
779    if(pos <= 1558){excess = 0;}
780    else {excess = 12000;}
781
782    b[0] = excess;
783
784      for(int i = 8; i <= 13; i++){
785      bitWrite(b[1],i-8,bitRead(excess,i));
786      }
787      bitSet(b[1],6);
788      SerialUSB.write(b,2);
789      bitClear(b[1],6);
790
791
792     b[2] = 0;
793
794     b[0] = amp;
795
796     for(int i = 8; i <= 13; i++){
797     bitWrite(b[1],i-8,bitRead(amp,i));
798     }
799
800     bitSet(b[1],7);
801     SerialUSB.write(b,3);
802     bitClear(b[1],7);
803
804     }
805
806
807
808  void send_func(){
809     b[0] = func;
810
811     for(int i = 8; i <= 13; i++){
812     bitWrite(b[1],i-8,bitRead(func,i));
813     }
814     bitClear(b[1],4);
815     bitClear(b[1],5);
```

```
816     b[2] = t;
817     SerialUSB.write(b,3);
818  }
819
820
821
822  void send_mu(){
823     b[0] = mu; //set byte b[0] equal to rightmost / lowest byte of 32 bit integer mu
824
825      for(int i = 8; i <= 13; i++){    // set byte b[1]'s bits # 0 to 5 equal
826     bitWrite(b[1],i-8,bitRead(mu,i)); // to bits 8 to 13 of 32 bit integer mu
827     }
828
829     bitSet(b[1],6);    // set byte b[1]'s bit # 6 to 1 to signal data of different kind
830     SerialUSB.write(b,2);  // bit-wise operators count bits from zero, like the
831     bitClear(b[1],6);      // powers of two they represent
832
833
834
835    }
836
837
838
839  void send_pos(){
840   b[0] = pos;
841
842     for(int i = 8; i <= 13; i++){
843     bitWrite(b[1],i-8,bitRead(pos,i));
844     }
845
846     //bitSet(b[1],6);
847     SerialUSB.write(b,2);
848     //bitClear(b[1],6);
849 }
850
851 void map_centre_back_to_pos(int last_pos, int try_num, int twos){
852   if( 10 > try_num && A0mu[last_pos] != centre){
853
854       if( A0mu[last_pos] < centre ){
855       try_num += 1; //start counting from 1 not 0 tries
856       twos *= 2; //avoid pow function as uses float
857       int new_pos = (last_pos + (1038/twos) ); //need powers of 2 fractions of half
    maximum
858       map_centre_back_to_pos( new_pos, try_num, twos );//recrusive call with advanced
    values
859         }
860       else if( A0mu[last_pos] > centre ){
861       try_num += 1; //start counting from 1 not 0 tries
862       twos *= 2; //avoid pow function as uses float
863       int new_pos = (last_pos - (1038/twos) ); //need powers of 2 fractions of half
    maximum
864       map_centre_back_to_pos( new_pos, try_num, twos );//recrusive call with advanced
    values
865         }
866
867
868   }
869   else if( A0mu[last_pos] == centre ){pos_rec = (pos_0 + last_pos);}
870
871   else{ // try_num == 10, so 2^try_num == 1024 and int 2076 / 1024 == 2
872     int get_out = 0;
873     for(int j = 1; j < 128; j++){//only widen value search range once entry range has
    been searched
874        for(int i = last_pos; i <= (last_pos + 2); i++){ // last_pos + 2076/2^try_num

875      if( A0mu[i] == centre + j ){pos_rec = (pos_0 + i); get_out = 1;break;}//break out i
    -loop
```

```
876       else if( A0mu[i] == centre - j ){pos_rec = (pos_0 + i); get_out = 1;break;}
877       }
878       if( 1 == get_out ){break;}//break out j-loop
879     } //in case no match is found
880     if( 0 == get_out ){// though unlikely, check entries 0, 1 and 2 as well
881       for(int j = 1; j < 128; j++){
882         for(int i = 0; i <= 2; i++){
883       if( A0mu[i] == centre + j ){pos_rec = (pos_0 + i); get_out = 1;break;}//break out i
    -loop
884       else if( A0mu[i] == centre - j ){pos_rec = (pos_0 + i); get_out = 1;break;}
885       }
886       if( 1 == get_out ){break;}//break out j-loop
887     }
888    }
889   }
890 }
891
892
```