Supplementary materials

Contents

1. Video descriptions

Supplementary Movie 1

The experimental process of autonomous high-throughput AFM single-cell mechanical analysis on living adherent cells at 37℃ in the CO_2-independent Leibovitz's L-15 medium without operator intervention. A microsphere-modified AFM probe was used. The AFM script program was used to automatically capture the optical bright-field image which was processed in real-time by the deep learning image recognition algorithm deployed on a laptop connected to the AFM desktop, generating the positional relationships between AFM probe and the cells within the horizontal detection area (100×100 μm^2) of the AFM probe. The positional relationships were passed to the AFM script program, and the script program then controlled the AFM probe to automatically perform force measurements on cells within the detection area of the AFM probe. Subsequently, the script program automatically moved the AFM sample stage to make the AFM probe reach a new position, captured the optical bright-field image and repeated the above process. The whole process was autonomous and did not require manual operation.

Supplementary Movie 2

The experimental process of autonomous high-throughput AFM single-cell mechanical analysis on living heterogeneous CTC cells at 37℃ in the L-15 medium without operator intervention.

Supplementary Movie 3

Autonomous high-throughput AFM single-cell indentation assay on co-cultured HaCaT cells and HMrSV5 cells. The detection area (100×100 μm^2) of the AFM probe is denoted by the yellow square, and the microspherical position of the AFM probe is denoted by the yellow diamond block. Recognition results of cell nuclei within the detection area are shown. The inset shows the force curves acquired on each probed cell.

Supplementary Movie 4

Autonomous high-throughput AFM single-cell indentation assay on co-cultured HaCaT cells and MCF-7 cells.

Supplementary Movie 5

Autonomous high-throughput AFM single-cell indentation assay on co-cultured HaCaT cells and MGC-803 cells.

Supplementary Movie 6

Collecting mixed CTCs (HaCaT cells and MCF-7cells) from blood by the contraction-expansion microfluidics. HaCaT cells (suspended) and MCF-7 cells (suspended) were added to the blood, which was then driven to pass through the contraction-expansion microchannel. Both HaCaT cells and MCF-7 cells were labeled with red fluorescein for visual verification.

Supplementary Movie 7

Autonomous high-throughput AFM single-cell indentation assay on mixed CTCs (HaCaT and MCF-7 cells) isolated from blood.

Supplementary Movie 8

Collecting mixed CTCs containing MCF-7 cells (suspended) and Raji cells from blood by the contraction-expansion microfluidics. Both MCF-7 cells and Raji cells were labeled with green fluorescein.

Supplementary Movie 9

Autonomous high-throughput AFM single-cell indentation assay on mixed CTCs (MCF-7 and Raji cells) isolated from blood.

2. Supplementary figures

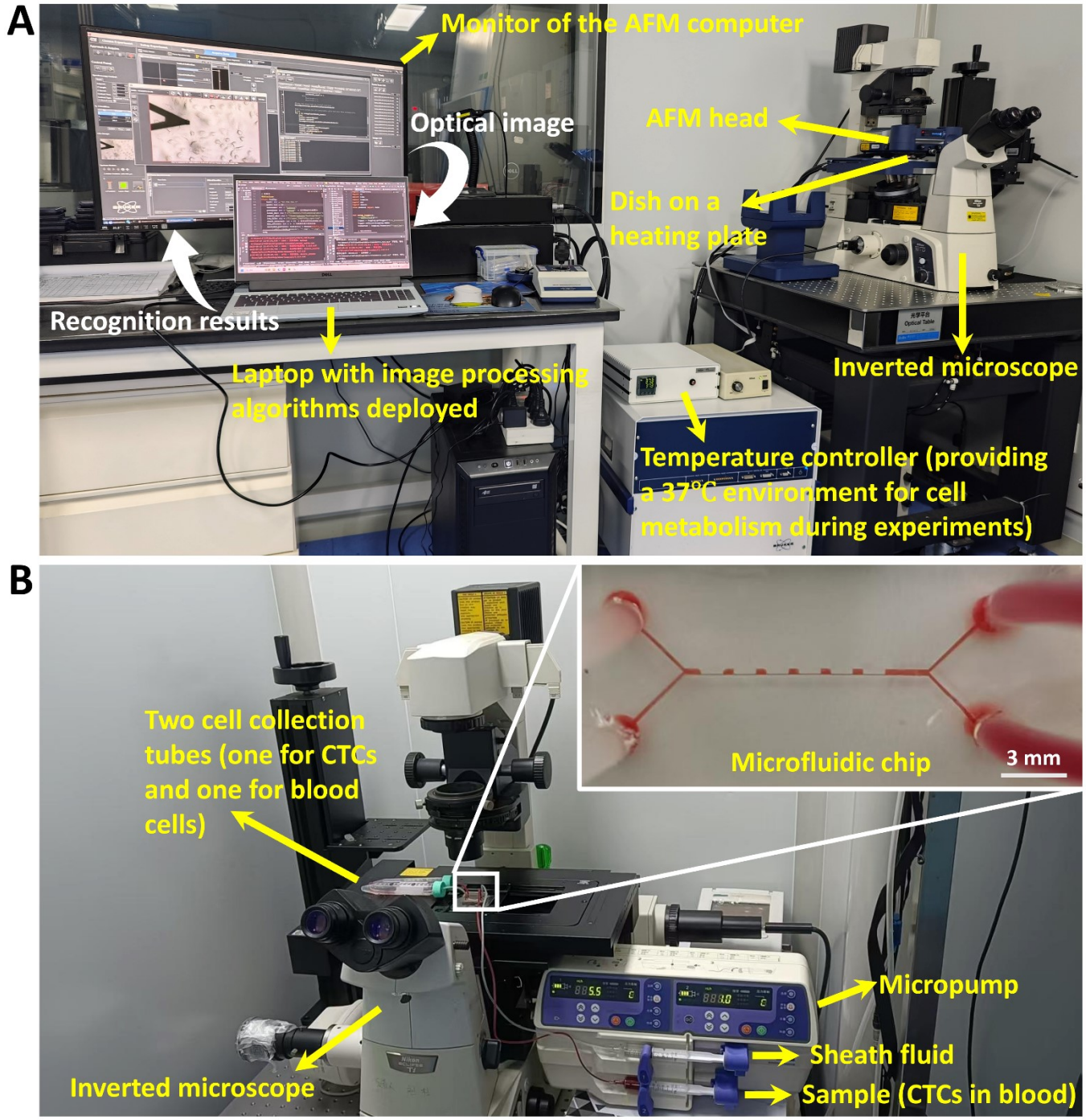

Figure S1 Experimental platform of autonomous high-throughput AFM single-cell indentation assay for revealing the mechanical signatures of mixed CTCs isolated from blood. (A) The autonomous single-cell mechanical measurement AFM system. The AFM is mounted on an inverted microscope, and the inverted microscope is used to capture the optical bright-field images of the AFM probe and cells during the experiment. The AFM has a heating plate, which provides the temperature environment (37°C) required for cell physiological activities, so AFM force measurements can be performed in the native states of living cells. A laptop computer deployed with deep learning image recognition algorithms is connected to the AFM computer. Cells and the AFM probe are recognized in the captured optical bright-field images by the deep learning algorithms, which are transmitted back to the AFM computer to guide automated force measurements on the recognized cells, and the process is repeated in a cycle without manual involvement. (B) The contraction-expansion microfluidic system for label-free isolation of CTCs from blood. The inset is the actual photograph of a contraction-expansion microfluidic chip (filled with blood to show the contraction-expansion microchannel). The dual-channel syringe pump is used to simultaneously inject the sheath fluid (PBS) and the blood sample containing CTCs. The microfluidic chip system is mounted on an inverted microscope to observe the sorting process.

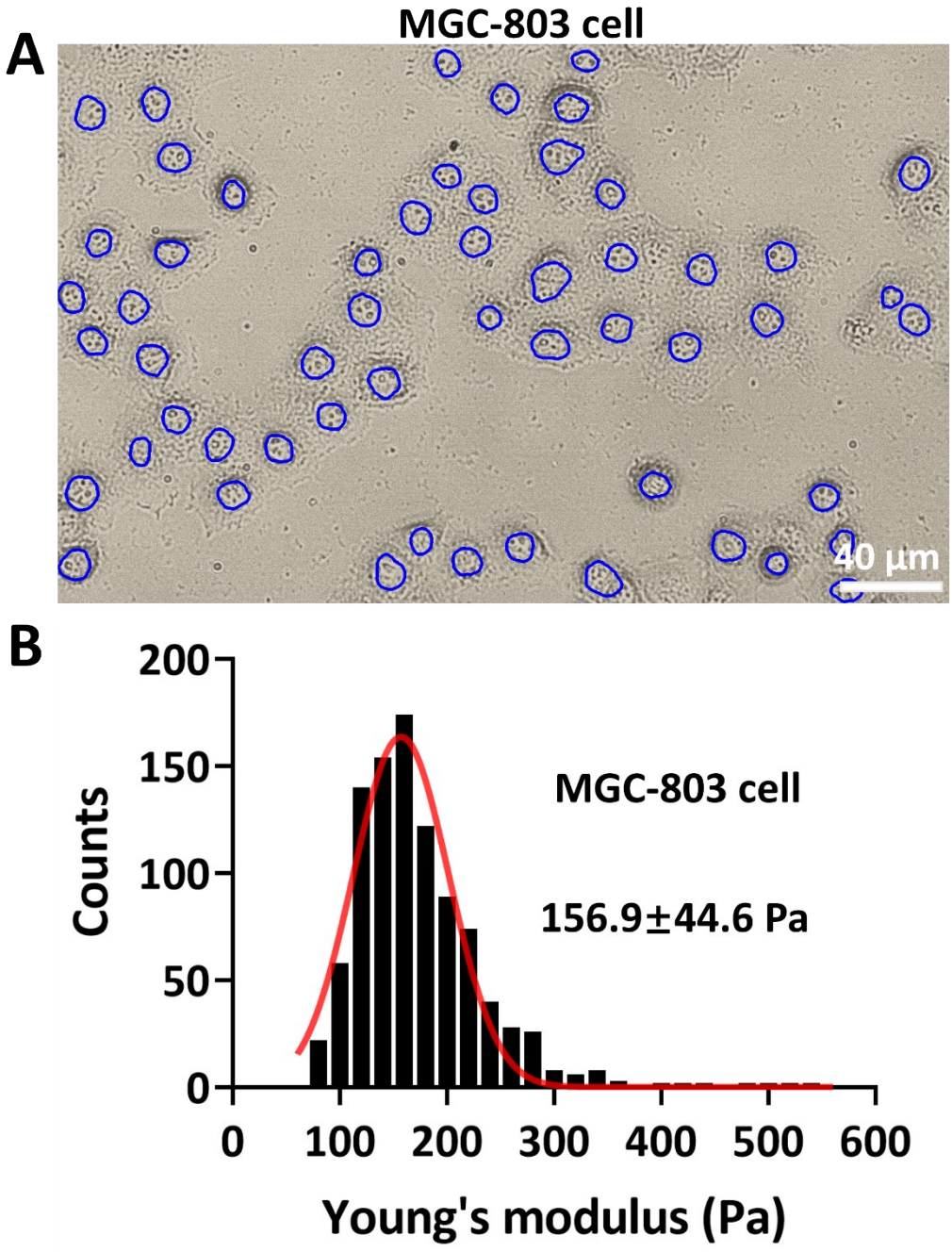

Figure S2 Experimental results on MGC-803 cells cultured alone. (A) Recognition results of the nuclei regions of MGC-803 cells in the optical bright-field images by the trained deep learning image recognition model. The blue curve indicates the outline of the recognized nucleus. (B) Statistical results of the Young's modulus of MGC-803cells cultured alone as reference values (200 cells were measured). The red curve represents the Gaussian distribution fitting results.

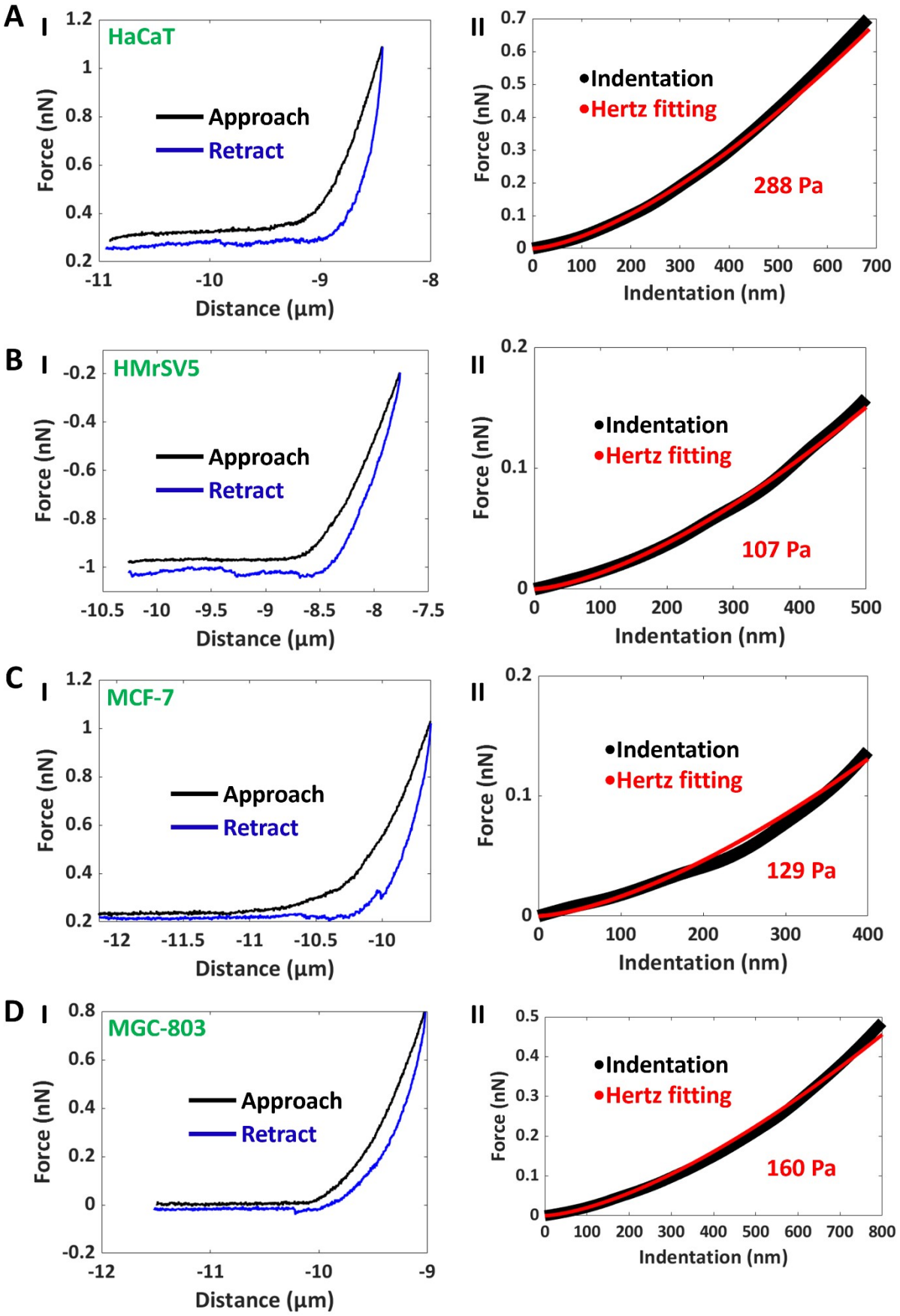

Figure S3 Typical force curves obtained on the four types of adherent cells cultured alone. (A) HaCaT cells. (B) HMrSV5 cells. (C) MCF-7 cells. (D) MGC-803 cells. (I) Force curves and (II) the corresponding fitting results of the indentation curves by Hertz model (to obtain cell Young's modulus).

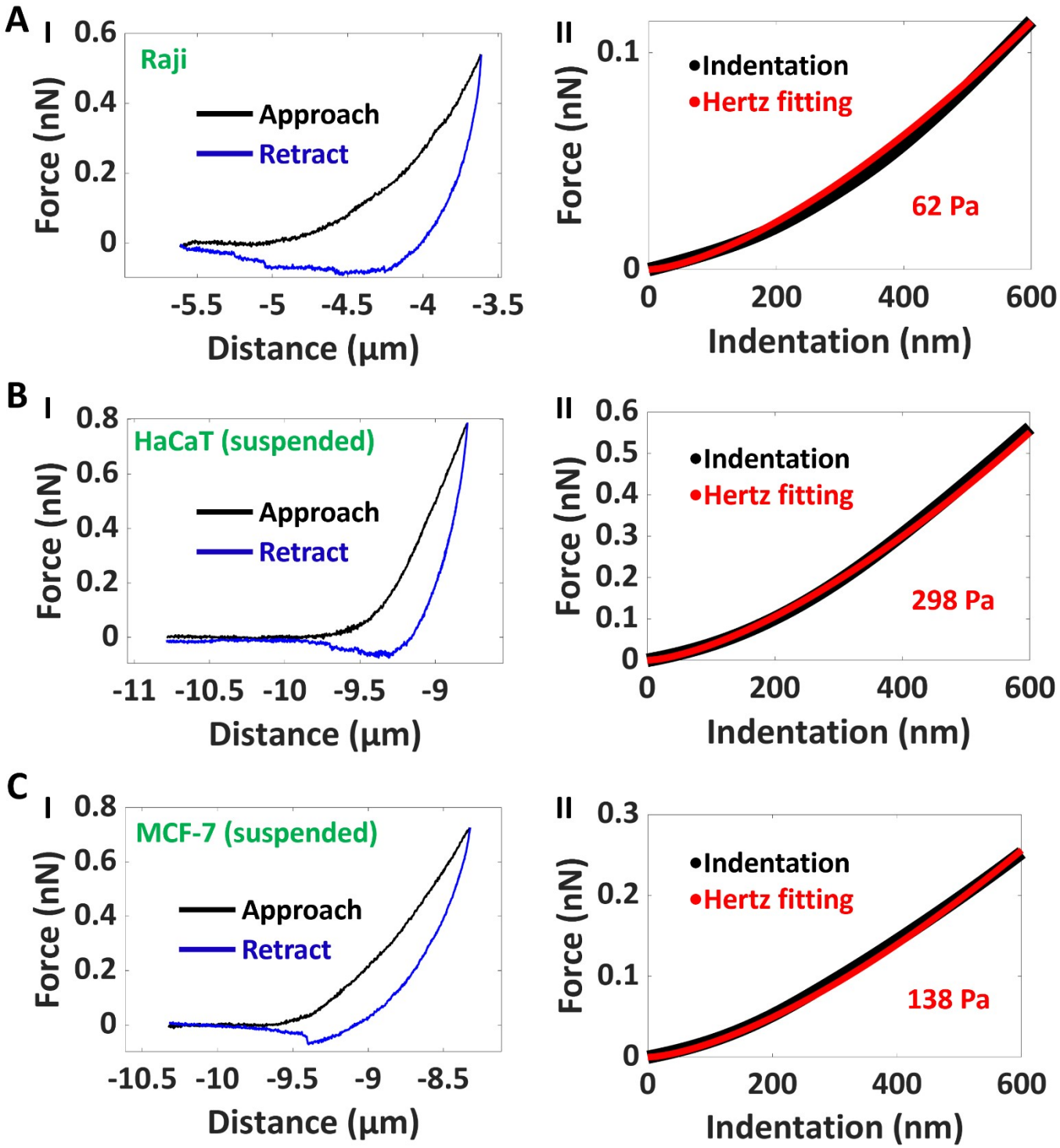

Figure S4 Typical force curves obtained on the three types of suspended cells cultured alone. (A) Raji cells. (B) Suspended HaCaT cells. (C) Suspended MCF-7 cells. (I) Force curves and (II) the corresponding fitting results of the indentation curves by Hertz model.

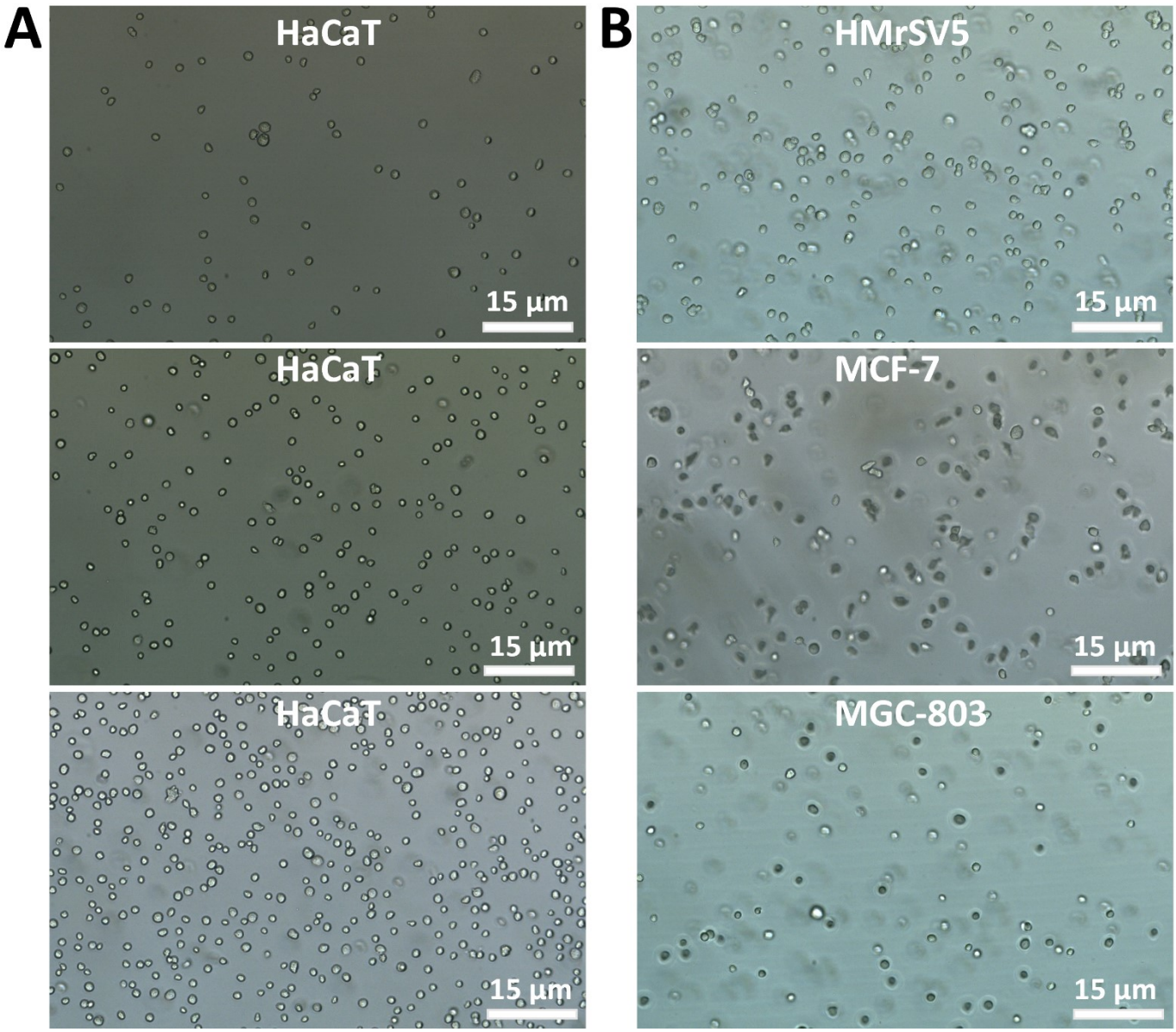

Figure S5 Optical bright-field images of digested adherent cells showing preparing the sample of co-cultured cells with different cell mixing ratios. (A) Digested HaCaT cells with different densities. (B) Digested HMrSV5 cells, MCF-7 cells and MGC-803 cells for adjusting cell densities.

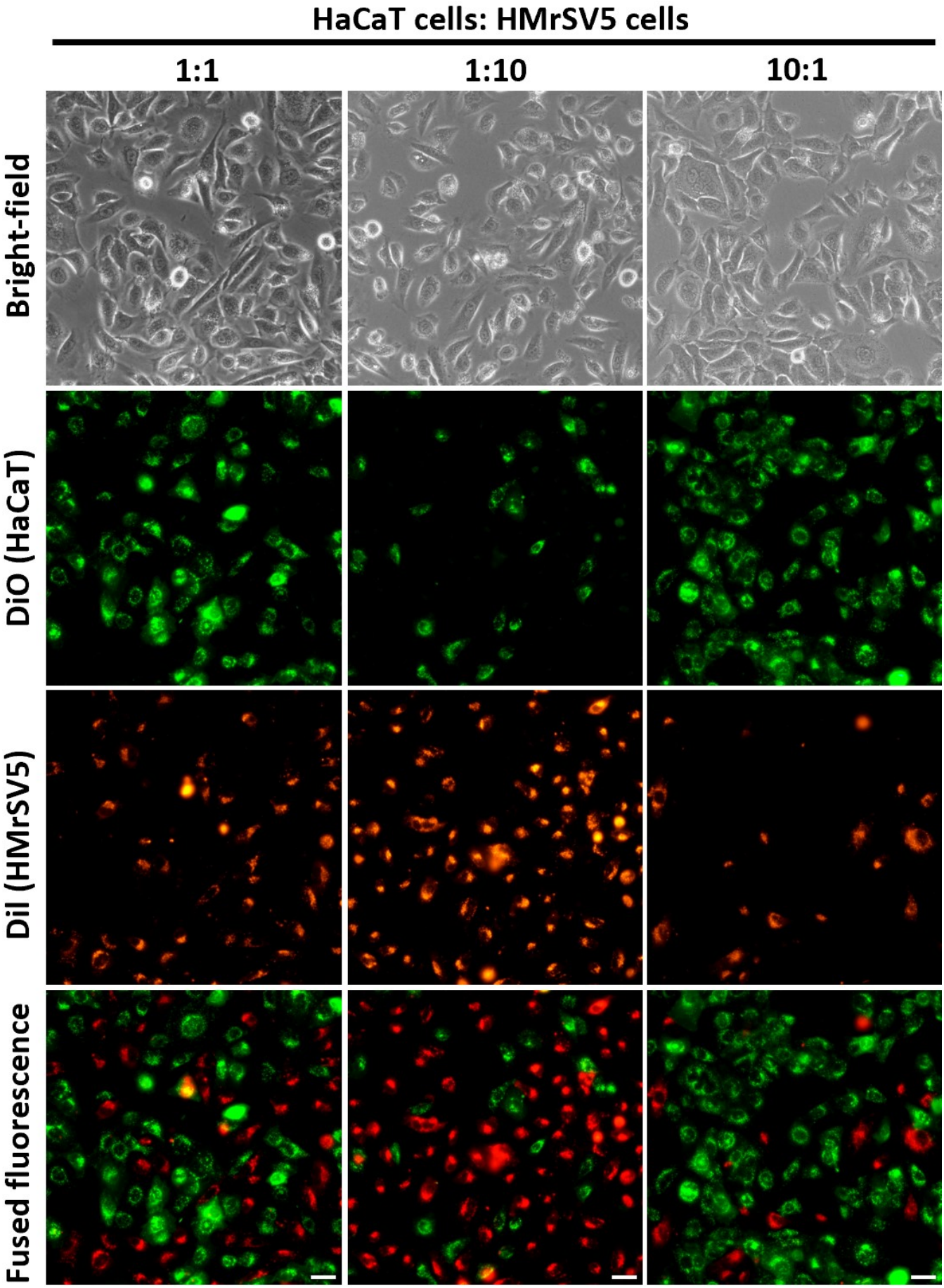

Figure S6 Fluorescence staining experiments of the co-culture of HaCaT cells and HMrSV5 cells with different cell mixing ratios. HaCaT cells were stained with green fluorescence, and HMrSV5 cells were stained with red fluorescence. The scale bar is 50 μm.

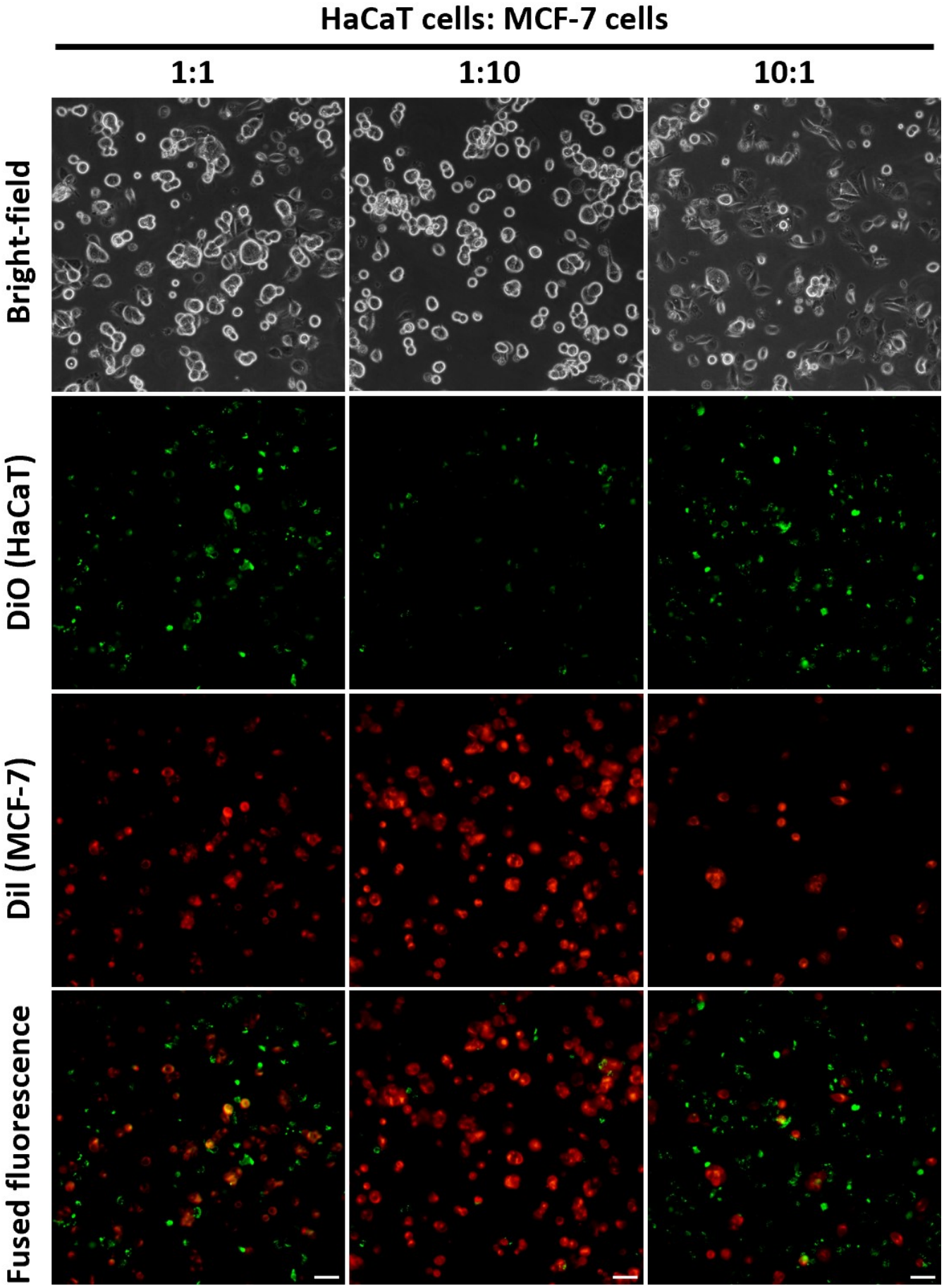

Figure S7 Fluorescence staining experiments of the co-culture of HaCaT cells and MCF-7 cells with different cell mixing ratios. HaCaT cells were stained with green fluorescence, and MCF-7 cells were stained with red fluorescence. The scale bar is 50 μm.

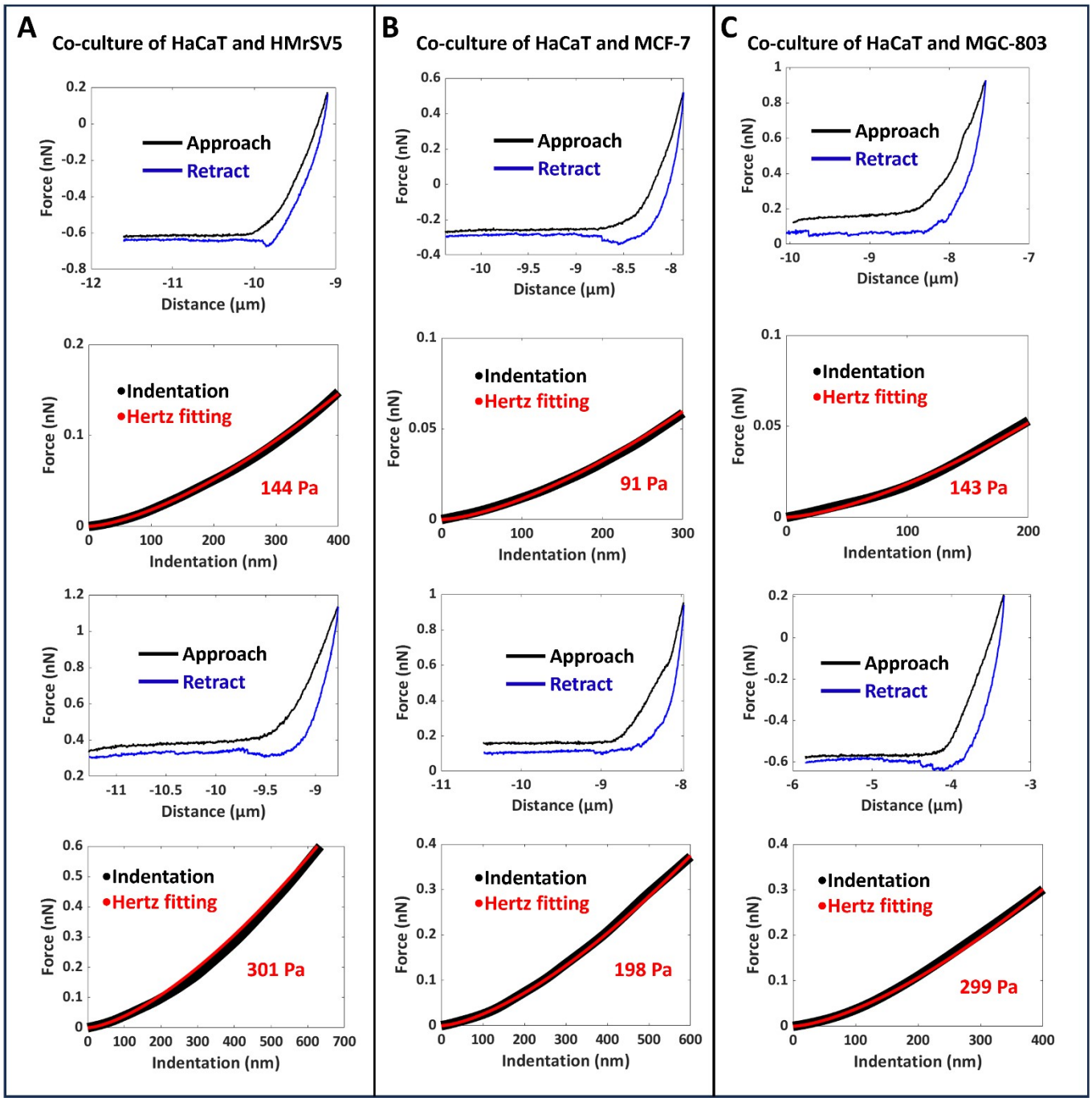

Figure S8 Typical force curves obtained during the autonomous high-throughput AFM single-cell indentation assay on co-cultured adherent cells. (A) Co-culture of HaCaT cells and HMrSV5 cells. (B) Co-culture of HaCaT cells and MCF-7 cells. (C) Co-culture of HaCaT cells and MGC-803 cells. For each co-culture condition, two representative force curves (corresponding to the two co-cultured cell types) and the Hertz fitting results are shown.

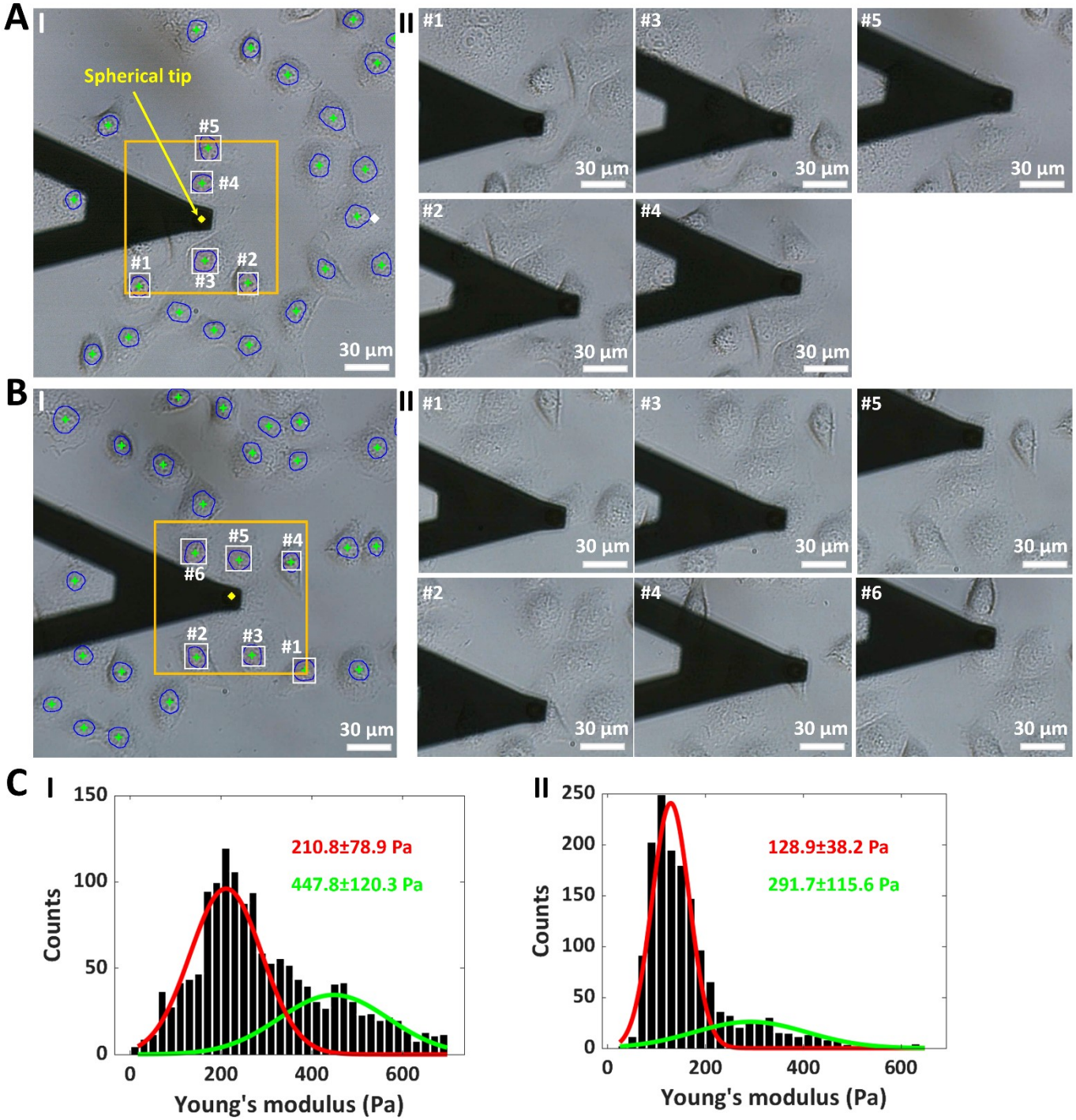

Figure S9 Experimental results of high-throughput AFM single-cell indentation assay on co-cultured HaCaT and MGC-803 cells. (A, B) Optical bright-field images showing the experimental process of two consecutive detection areas. (I) Recognition results of the optical bright-field image of AFM probe and cells. The orange box indicates the detection area, and the yellow diamond block indicates the recognized AFM microspherical tip. The outline of the recognized cell nucleus is indicated by a blue curve and the center of the nucleus is indicated by a green cross. The white diamond block indicates the center of the next detection area. (II) The AFM probe has been accurately moved to each recognized cell within the detection area to perform indentation assay one by one. Five cells were recognized and measured (denoted by the symbols #1-#5) in the detection area in (A), and six cells were recognized and measured in the detection area (denoted by the symbols #1-#6) in (B). (C) Statistical results of the constructed Young's modulus profiles of the co-cultured cells with different cell mixing ratios. Double-Gaussian fittings were performed (red and green curve respectively). (I-II) HaCaT cells and MGC-803 cells were mixed at a ratio of 10:1 (I) and 1:1 (II) respectively. For each co-culture condition (cell mixing ratio of 10:1 and 1:1), 300 cells were measured.

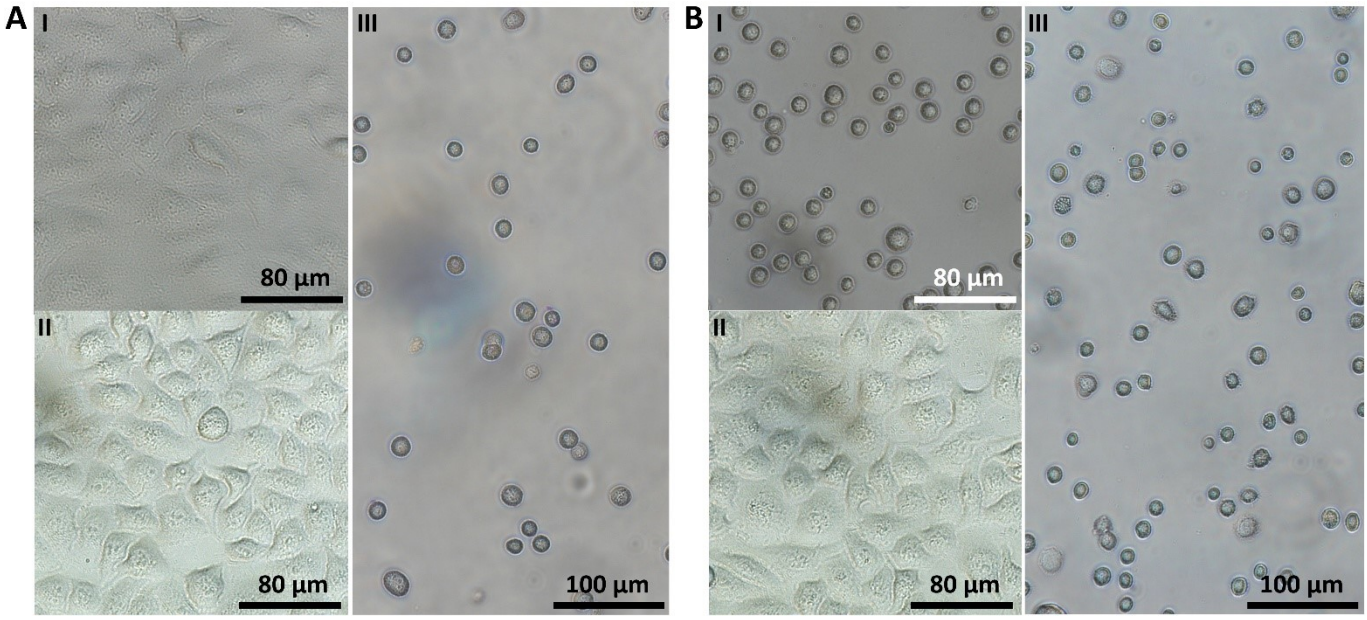

Figure S10 Optical bright-field images showing preparing the sample of mixed CTCs. (A) Mixed HaCaT cells (suspended) and MCF-7 cells (suspended). (I) The cultured HaCaT cells. (II) The cultured MCF-7 cells. (III) Mixture of the digested HaCaT cells and digested MCF-7 cells. (B) Mixed MCF-7 cells (suspended) and Raji cells. (I) The cultured Raji cells. (II) The cultured MCF-7 cells. (III) Mixture of Raji cells and digested MCF-7 cells.

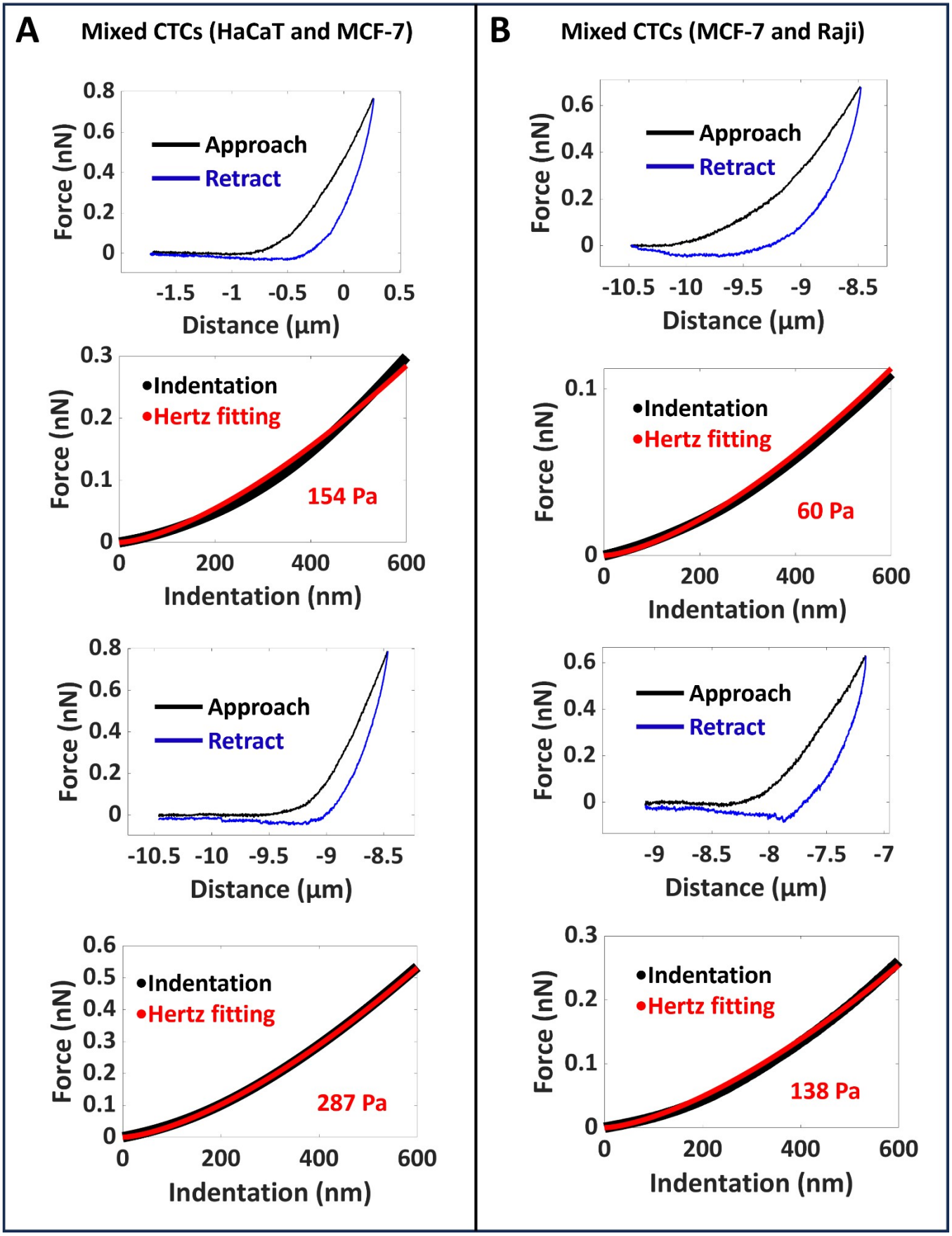

Figure S11 Typical force curves obtained during the autonomous high-throughput AFM single-cell indentation assay on mixed CTCs isolated from blood. (A) Mixed HaCaT cells (suspended) and MCF-7 cells (suspended). (B) Mixed MCF-7 cells (suspended) and Raji cells. For each condition, two representative force curves (corresponding to the two CTC types) and the Hertz fitting results are shown.

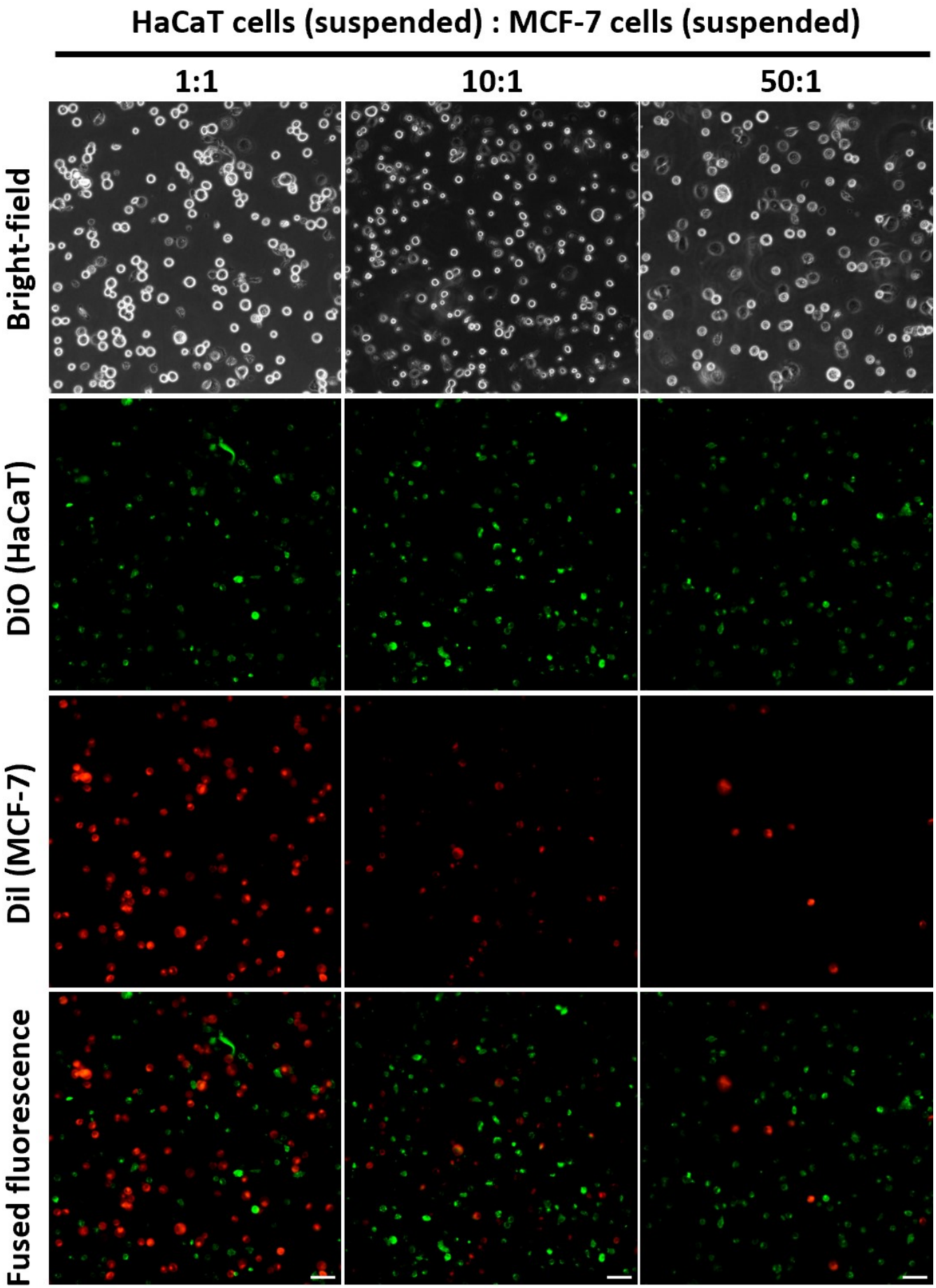

Figure S12 Fluorescence staining experiments of the mixed HaCaT cells (suspended) and MCF-7 cells (suspended) with different cell mixing ratios. HaCaT cells were stained with green fluorescence, and MCF-7 cells were stained with red fluorescence. The scale bar is 50 μm.

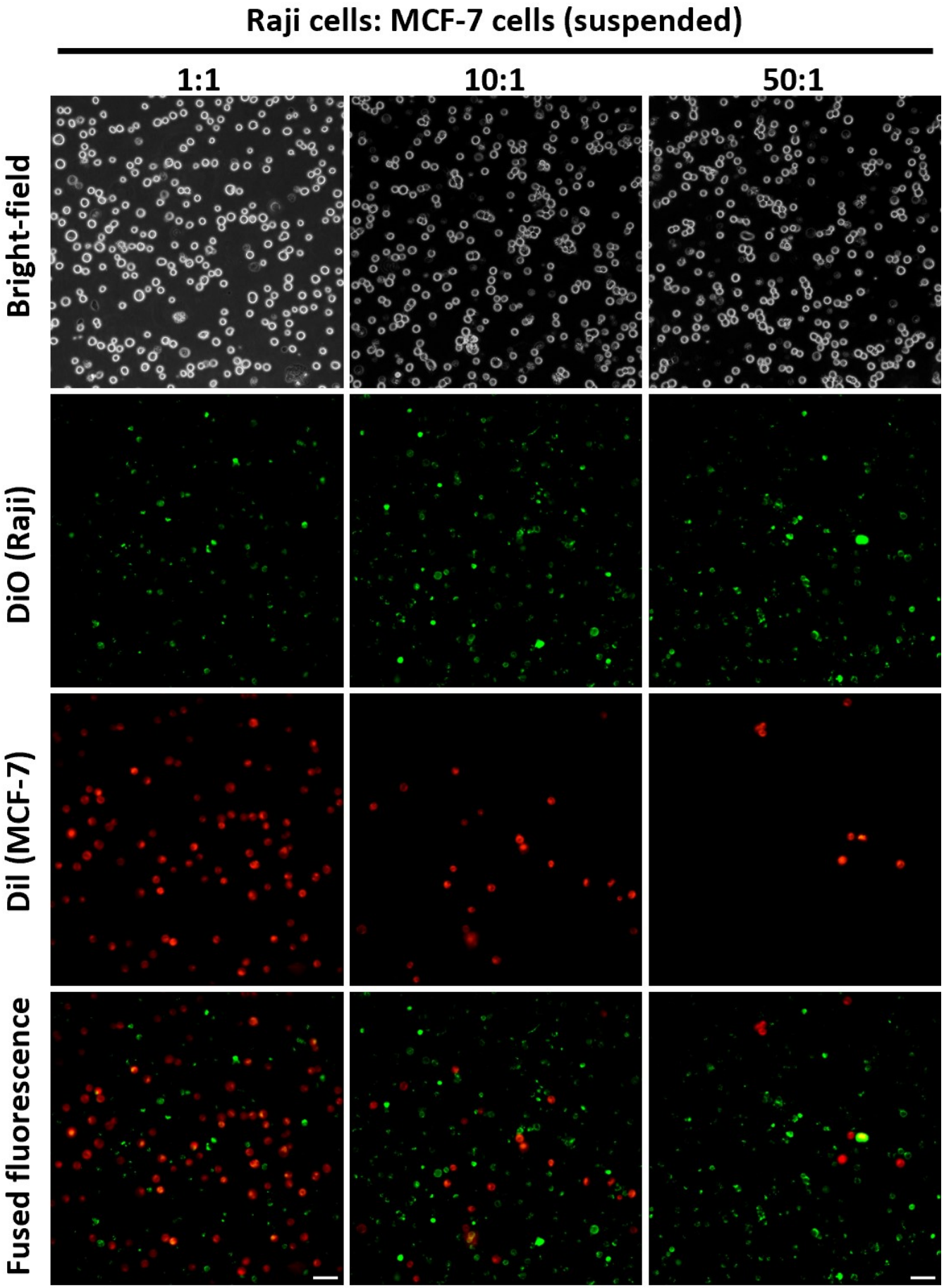

Figure S13 Fluorescence staining experiments of the mixed Raji cells and MCF-7 cells (suspended) with different cell mixing ratios. Raji cells were stained with green fluorescence, and MCF-7 cells were stained with red fluorescence. The scale bar is 50 μm.

3. Source codes

The source codes used in the work have been uploaded to GitHub website and are publicly available. Researchers can freely download the source codes from the GitHub, including autonomous high-throughput AFM single-cell mechanical analysis on adherent cells (https://github.com/xxxxrrrrxx/Auto-AFM/tree/main/Pyramid-Unet) and autonomous high-throughput AFM single-cell mechanical analysis on CTC cells (https://github.com/cnncell/Combination-of-YOLOv7-and-U-Net). The instruction documentations and the main codes are as follows.

3.1 Autonomous high-throughput AFM single-cell mechanical analysis on adherent cells

3.1.1 Instruction documentation

(1) Model Training

①Dataset Preparation: Collect bright-field cell images with diverse brightness and contrast captured by JPK to ensure data diversity and representativeness, providing rich samples for subsequent training.

②Labeling: Use the professional labeling tool LabelMe for pixel-level annotation of original images, defining the boundaries and category information of cell nuclei to generate label images consistent with the original dimensions.

③Pyramid-UNet Construction: Build the Pyramid-UNet network structure based on PyTorch. Set parameters such as network layers, channel numbers, and convolution kernel size in net.py, and define input/output formats. data.py ensures one-to-one correspondence between original images and labels, while utils.py standardizes image sizes during training.

④Model Training: Input the training set into the Pyramid-UNet model and run train.py for training. Calculate prediction results via forward propagation, compute loss against labels, and update network parameters via backpropagation. Regularly evaluate model performance using a validation set during training, adjusting hyperparameters (e.g., learning rate, batch size) based on validation results.

⑤Image Segmentation: Run predict.py to perform semantic segmentation on cell nucleus images using the Pyramid-UNet model. After cropping and resizing, extract contours and mark nucleus boundaries, then convert to a binary image and call external scripts for fine-grained analysis. This is followed by template_matching.py, which locates probe tips via template matching, extracts nucleus centroid coordinates, defines a 250×250 pixel detection range centered on the probe, calculates the relative distance from nuclei to the probe (converted by a coefficient of -0.02e-5), and saves results to a CSV file with visual annotations.

(2) Automated AFM Experiments

①Run Run.py to monitor and process input image files in specified paths. Call the prediction script predict.py to generate output files (output_table.csv), clean temporary files, and delete original inputs after processing. It supports cyclic processing, error retry, and logs via a logger.

②Run autosyn.py, an SFTP-based bidirectional file synchronization program that monitors local file system changes via Watchdog and periodically checks remote server file status to achieve automatic synchronization. The program supports operations like upload, download, and deletion, with retry mechanisms and error handling.

③Run JPK.py within the JPK software to real-time acquire mechanical curves of cell nuclei within the detection range.

(3) Additional Notes

Both synchronization and neural network codes run on the PyCharm client.

The synchronization codes autosyn.py and Run.py run simultaneously.

File synchronization uses WinSCP for data transfer via SFTP.

References

https://github.com/xxxr9802/Pyramid-UNet

https://github.com/qiaofengsheng/pytorch-UNet.git

3.1.2 AFM script for automated control of the AFM probe and the sample stage

```
# Python code for ExperimentPlanner

import time
import os
import csv
checkVersion('SPM', 7, 0, 178)

file_path = '/home/jpkuser/Desktop/data-transmit/output_table.csv'
filename = '/home/jpkuser/Desktop/data-transmit/1.tif'
count = 0
def wait_for_file_path(file_path):
    while not os.path.exists(file_path):
        time.sleep(2)

Snapshooter.saveOpticalSnapshot(filename)
n = 0
while 1:
        #Coordinates of the next area
        nextAreaX = 0
        nextAreaY = -10e-5



        #Number of detectable cells in the area
        coordinate_count = 0
        #file_path = '/home/jpkuser/Desktop/data-transmit/output_table.csv'
        wait_for_file_path(file_path)
        # If the file path exists, continue to execute the subsequent code

        #Read coordinate points within probe detection range into script
        def addposition(x,y):
            ForceSpectroscopy.addPosition(x, y)

        #Disabled platform moves
        MotorizedStage.disengage()

        #Clear coordinates, read new coordinates
        ForceSpectroscopy.clearPositions()
        #init

        #Add initial position to software table, set to origin, index 0
```

```
ForceSpectroscopy.addPosition(0, 0)

#Read the coordinates of the detectable point and the center point of the next area into the script
with open(file_path, mode='r') as file:
    reader = csv.reader(file)
    next(reader)
    for row in reader:
        x = float(row[0])
        y = float(row[1])
        if abs(x) < 5e-5 and abs(y) < 5e-5 :
            addposition(x, y)
            coordinate_count += 1
        #else :
            #nextAreaX = x
            #nextAreaY = y

#Probe tip moves back to initial position
ForceSpectroscopy.moveToForcePositionIndex(0)

#Get force curves for all detectable cells and save them automatically
i = 0
for j in xrange(coordinate_count):
    i+=1
    Scanner.retractPiezo()
    Scanner.retract()
    ForceSpectroscopy.moveToForcePositionIndex(i)

    Scanner.approach()
    ForceSpectroscopy.startScanning(5)
    Scanner.retractPiezo()
    Scanner.moveMotorsUp(2e-5)
    time.sleep(1.0)

#After detecting each area, the probe must return to its original position
ForceSpectroscopy.moveToForcePositionIndex(0)

#Enabling the platform
MotorizedStage.engage()

#Move the platform to the next regional center
MotorizedStage.moveToRelativePosition(nextAreaX, nextAreaY)
```

```
            n+=1
            #Disabled platform
            MotorizedStage.disengage()

            #Probe down and up
            Scanner.approach()

            time.sleep(1.0)
            Scanner.retract()
            os.remove(file_path)
            time.sleep(2.0)
            Snapshooter.saveOpticalSnapshot(filename)
            time.sleep(2.0)
            print("wait new document.....",n)
```

3.1.3 Synchronous real-time data transmission (optical bright-field images and recognition results)

```
import os
import time
import logging
import threading
import paramiko
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
from typing import Dict, List, Optional, Set, Tuple
from dataclasses import dataclass, field
from enum import Enum

# Operation Type Enumeration
class OperationType(Enum):
    UPLOAD = "upload"
    DOWNLOAD = "download"
    DELETE_LOCAL = "delete_local"
    DELETE_REMOTE = "delete_remote"



# Configuration Class
@dataclass
class Config:
    host: str = "10.254.254.1"
    port: int = 22
    username: str = "jpkuser"
    password: str = "jpkjpk"    # Password hardcoded in configuration
```

```
    local_dir: str = r"C:\Users\17105\Desktop\data-transmit"
    remote_dir: str = "/home/jpkuser/Desktop/data-transmit"
    log_file: str = r"C:\sync_logs\sftp_sync.log"
    check_interval: int = 10    # Interval (seconds) for checking remote changes
    max_retries: int = 3    # Maximum retries for failed operations
    excluded_patterns: List[str] = field(default_factory=lambda: [
        '.git', 'node_modules', '*.tmp', '*.log', 'logs/'
    ])


# File Status Class
@dataclass
class FileState:
    mtime: float
    size: int
    exists: bool
    last_checked: float = field(default_factory=time.time)


# SFTP Connection Manager
class SftpConnector:
    def __init__(self, config: Config, logger: logging.Logger):
        self.config = config
        self.logger = logger
        self.ssh = None
        self.sftp = None
        self.connected = False

    def connect(self) -> bool:
        """Establish SFTP connection"""
        try:
            if self.connected:
                self.disconnect()

            self.ssh = paramiko.SSHClient()
            self.ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
            self.ssh.connect(
                self.config.host,
                port=self.config.port,
                username=self.config.username,
                password=self.config.password,
```

```
                timeout=10
            )

            self.sftp = self.ssh.open_sftp()
            self.connected = True
            self.logger.info("SFTP connection established successfully")
            return True
        except Exception as e:
            self.logger.error(f"Failed to establish SFTP connection: {e}")
            self.connected = False
            return False

    def disconnect(self):
        """Disconnect SFTP connection"""
        try:
            if self.sftp:
                self.sftp.close()
            if self.ssh:
                self.ssh.close()
            self.connected = False
            self.logger.info("SFTP connection disconnected")
        except Exception as e:
            self.logger.error(f"Error disconnecting SFTP connection: {e}")

    def execute_with_retry(self, func, *args, **kwargs):
        """SFTP operation executor with retry mechanism"""
        retries = 0
        while retries < self.config.max_retries:
            try:
                if not self.connected and not self.connect():
                    retries += 1
                    time.sleep(2)
                    continue
                return func(*args, **kwargs)
            except FileNotFoundError as e:
                self.logger.warning(f"File not found: {e}")
                self.connected = False
                return False    # File not found, no need to retry
            except Exception as e:
                self.logger.error(f"SFTP operation failed: {e}")
                self.connected = False
```

```
                retries += 1
                time.sleep(2)
        self.logger.error(f"Operation reached maximum retries: {func.__name__}")
        return None

    def file_exists(self, remote_path: str) -> bool:
        """Check if remote file exists"""

        def _check():
            try:
                self.sftp.stat(remote_path)
                return True
            except FileNotFoundError:
                return False

        return self.execute_with_retry(_check)

    def get_remote_file_list(self) -> Optional[Dict[str, Dict]]:
        """Get list of files in remote directory"""

        def _list_files():
            files = {}
            try:
                for item in self.sftp.listdir_attr(self.config.remote_dir):
                    if item.filename in ['.', '..']:
                        continue
                    files[item.filename] = {
                        'size': item.st_size,
                        'mtime': item.st_mtime,
                        'is_dir': item.st_mode & 0o40000 != 0
                    }
            except Exception as e:
                self.logger.error(f"Failed to get remote file list: {e}")
                return None
            return files

        return self.execute_with_retry(_list_files)

    def upload_file(self, local_path: str, remote_path: str) -> bool:
        """Upload file to remote"""
```

```
        def _upload():
            # Ensure target directory exists
            remote_dir = os.path.dirname(remote_path)
            self._ensure_remote_dir_exists(remote_dir)

            # Upload file
            self.sftp.put(local_path, remote_path)
            return True

        return self.execute_with_retry(_upload)

    def download_file(self, remote_path: str, local_path: str) -> bool:
        """Download file from remote"""

        def _download():
            # Check if remote file exists
            if not self.file_exists(remote_path):
                self.logger.warning(f"Remote file does not exist, skipping download: {remote_path}")
                return False

            # Ensure local directory exists
            local_dir = os.path.dirname(local_path)
            if not os.path.exists(local_dir):
                os.makedirs(local_dir)

            # Download file
            self.sftp.get(remote_path, local_path)
            return True

        return self.execute_with_retry(_download)

    def delete_remote_file(self, remote_path: str) -> bool:
        """Delete remote file"""

        def _delete():
            # Check if remote file exists
            if not self.file_exists(remote_path):
                self.logger.warning(f"Remote file does not exist, skipping deletion: {remote_path}")
                return True

            # Delete file
```

```
                self.sftp.remove(remote_path)
                return True

            return self.execute_with_retry(_delete)

        def _ensure_remote_dir_exists(self, remote_dir: str):
            """Ensure remote directory exists"""
            try:
                self.sftp.stat(remote_dir)
            except FileNotFoundError:
                parent_dir = os.path.dirname(remote_dir)
                if parent_dir != remote_dir:
                    self._ensure_remote_dir_exists(parent_dir)
                self.sftp.mkdir(remote_dir)


    # Local File System Event Handler
    class LocalSyncHandler(FileSystemEventHandler):
        def __init__(self, sync_manager):
            self.sync_manager = sync_manager

        def on_any_event(self, event):
            """Handle file system events"""
            if event.is_directory or self.sync_manager.is_excluded(event.src_path):
                return

            # Brief delay to avoid incomplete file operations
            time.sleep(0.1)

            # Calculate relative path and remote path
            rel_path = os.path.relpath(event.src_path, self.sync_manager.config.local_dir)
            remote_path = os.path.join(self.sync_manager.config.remote_dir, rel_path).replace('\\', '/')

            if event.event_type == 'created' or event.event_type == 'modified':
                self.sync_manager.logger.info(f"Local creation/modification detected: {rel_path}")
                self.sync_manager.perform_sync(OperationType.UPLOAD, event.src_path, remote_path)

            elif event.event_type == 'deleted':
                self.sync_manager.logger.info(f"Local deletion detected: {rel_path}")
                self.sync_manager.perform_sync(OperationType.DELETE_REMOTE, None, remote_path)
```

```
# File Synchronization Manager
class SyncManager:
    def __init__(self, config: Config):
        self.config = config
        self.logger = self._setup_logger()
        self.sftp = SftpConnector(config, self.logger)
        self.running = False
        self.observer = None
        self.remote_watcher_thread = None
        self.sync_lock = threading.Lock()    # Synchronization lock
        self.local_state: Dict[str, FileState] = {}    # Local file state
        self.remote_state: Dict[str, FileState] = {}    # Remote file state

    def _setup_logger(self):
        """Setup logger"""
        logger = logging.getLogger("sftp_sync")
        logger.setLevel(logging.INFO)

        if not logger.handlers:
            # Ensure log directory exists
            log_dir = os.path.dirname(self.config.log_file)
            if not os.path.exists(log_dir):
                os.makedirs(log_dir)

            file_handler = logging.FileHandler(self.config.log_file, encoding='utf-8')
            console_handler = logging.StreamHandler()
            formatter = logging.Formatter('%(asctime)s - %(levelname)s - %(message)s')
            file_handler.setFormatter(formatter)
            console_handler.setFormatter(formatter)
            logger.addHandler(file_handler)
            logger.addHandler(console_handler)

        return logger

    def is_excluded(self, path: str) -> bool:
        """Check if path should be excluded"""
        for pattern in self.config.excluded_patterns:
            if pattern.startswith('*.') and path.endswith(pattern[1:]):
                return True
            if pattern in path:
```

```
                return True
        return False

    def start(self):
        """Start synchronization service"""
        self.logger.info("Starting file synchronization service...")
        self.running = True

        self.initial_sync()
        self._start_local_watcher()
        self._start_remote_watcher()

        self.logger.info("File synchronization service started")

    def stop(self):
        """Stop synchronization service"""
        self.logger.info("Stopping file synchronization service...")
        self.running = False

        if self.observer:
            self.observer.stop()
            self.observer.join(timeout=5)

        if self.remote_watcher_thread:
            self.remote_watcher_thread.join(timeout=5)

        self.sftp.disconnect()
        self.logger.info("File synchronization service stopped")

    def perform_sync(self, action: OperationType, local_path: Optional[str], remote_path: Optional[str]):
        """Perform synchronization operation"""
        self.logger.info(f"Performing synchronization operation: {action.value} {local_path or remote_path}")

        with self.sync_lock:    # Ensure only one synchronization operation at a time
            success = False
            retries = 0

            while retries < self.config.max_retries:
                try:
                    if action == OperationType.UPLOAD:
                        if not os.path.exists(local_path):
```

```
                self.logger.warning(f"Local file does not exist, skipping upload: {local_path}")
                return False

            success = self.sftp.upload_file(local_path, remote_path)
            if success:
                # Update local and remote states
                rel_path = os.path.relpath(local_path, self.config.local_dir)
                try:
                    stat = os.stat(local_path)
                    self.local_state[rel_path] = FileState(
                        mtime=stat.st_mtime,
                        size=stat.st_size,
                        exists=True,
                        last_checked=time.time()
                    )
                    self.remote_state[rel_path] = FileState(
                        mtime=stat.st_mtime,
                        size=stat.st_size,
                        exists=True,
                        last_checked=time.time()
                    )
                except Exception as e:
                    self.logger.warning(f"Failed to update file state: {e}")

        elif action == OperationType.DOWNLOAD:
            success = self.sftp.download_file(remote_path, local_path)
            if success:
                # Update local and remote states
                rel_path = os.path.relpath(local_path, self.config.local_dir)
                try:
                    stat = os.stat(local_path)
                    self.local_state[rel_path] = FileState(
                        mtime=stat.st_mtime,
                        size=stat.st_size,
                        exists=True,
                        last_checked=time.time()
                    )
                    self.remote_state[rel_path] = FileState(
                        mtime=stat.st_mtime,
                        size=stat.st_size,
                        exists=True,
```

```
                        last_checked=time.time()
                    )
                except Exception as e:
                    self.logger.warning(f"Failed to update file state: {e}")

            elif action == OperationType.DELETE_REMOTE:
                # Calculate relative path
                rel_path = os.path.basename(remote_path) if remote_path else ""

                # Check if remote file exists
                if self.sftp.file_exists(remote_path):
                    success = self.sftp.delete_remote_file(remote_path)
                    if success:
                        # Update remote state immediately to prevent duplicate operations
                        self.remote_state[rel_path] = FileState(
                            mtime=0,
                            size=0,
                            exists=False,
                            last_checked=time.time()
                        )
                else:
                    success = True    # File does not exist, consider operation successful

            elif action == OperationType.DELETE_LOCAL:
                if os.path.exists(local_path):
                    try:
                        # Attempt to delete file
                        os.remove(local_path)
                        success = True
                        # Update local state
                        rel_path = os.path.relpath(local_path, self.config.local_dir)
                        self.local_state[rel_path] = FileState(
                            mtime=0,
                            size=0,
                            exists=False,
                            last_checked=time.time()
                        )
                    except PermissionError as e:
                        self.logger.warning(f"File is locked, cannot delete: {local_path}, Error: {e}")
                        success = False
                else:
```

```
                                success = True    # File does not exist, consider operation successful

                            if success:
                                self.logger.info(f"Synchronization successful: {action.value} {local_path or remote_path}")
                                break
                            else:
                                retries += 1
                                self.logger.warning(f"Synchronization failed, retrying ({retries}/{self.config.max_retries}): {action.value}")
                                time.sleep(1)

                        except Exception as e:
                            retries += 1
                            self.logger.error(f"Synchronization error, retrying ({retries}/{self.config.max_retries}): {e}")
                            time.sleep(1)

                    if not success:
                        self.logger.error(f"Final synchronization failure: {action.value} {local_path or remote_path}")

            def initial_sync(self):
                """Perform initial two-way synchronization"""
                self.logger.info("Performing initial two-way synchronization...")

                local_files = self._get_local_file_list()
                remote_files = self.sftp.get_remote_file_list()

                if not local_files or not remote_files:
                    self.logger.warning("Initial synchronization failed: Cannot retrieve file lists")
                    return

                # Compare files on both sides, use latest modification time as criterion
                for filename in set(local_files.keys()).union(set(remote_files.keys())):
                    local_info = local_files.get(filename)
                    remote_info = remote_files.get(filename)

                    if local_info and not remote_info:
                        # Local file exists, remote does not -> Upload
                        local_path = os.path.join(self.config.local_dir, filename)
                        remote_path = os.path.join(self.config.remote_dir, filename).replace('\\', '/')
                        self.perform_sync(OperationType.UPLOAD, local_path, remote_path)
```

```
            elif not local_info and remote_info:
                # Local file does not exist, remote does -> Download
                local_path = os.path.join(self.config.local_dir, filename)
                remote_path = os.path.join(self.config.remote_dir, filename).replace('\\', '/')
                self.perform_sync(OperationType.DOWNLOAD, local_path, remote_path)

            elif local_info and remote_info:
                # Both exist, compare modification times
                if local_info['mtime'] > remote_info['mtime']:
                    # Local is newer -> Upload
                    local_path = os.path.join(self.config.local_dir, filename)
                    remote_path = os.path.join(self.config.remote_dir, filename).replace('\\', '/')
                    self.perform_sync(OperationType.UPLOAD, local_path, remote_path)
                elif remote_info['mtime'] > local_info['mtime']:
                    # Remote is newer -> Download
                    local_path = os.path.join(self.config.local_dir, filename)
                    remote_path = os.path.join(self.config.remote_dir, filename).replace('\\', '/')
                    self.perform_sync(OperationType.DOWNLOAD, local_path, remote_path)

        # Update states
        self._update_file_states(local_files, remote_files)

    def _update_file_states(self, local_files: Dict, remote_files: Dict):
        """Update file states"""
        current_time = time.time()

        # Update local state
        for filename, info in local_files.items():
            self.local_state[filename] = FileState(
                mtime=info['mtime'],
                size=info['size'],
                exists=True,
                last_checked=current_time
            )

        # Handle deleted local files
        for filename in list(self.local_state.keys()):
            if filename not in local_files:
                if self.local_state[filename].exists:
                    self.local_state[filename] = FileState(
                        mtime=0,
```

```
                        size=0,
                        exists=False,
                        last_checked=current_time
                    )
                else:
                    # If already marked as non-existent and not checked for a long time, remove state
                    if current_time - self.local_state[filename].last_checked > 3600:
                        del self.local_state[filename]

        # Update remote state
        for filename, info in remote_files.items():
            self.remote_state[filename] = FileState(
                mtime=info['mtime'],
                size=info['size'],
                exists=True,
                last_checked=current_time
            )

        # Handle deleted remote files
        for filename in list(self.remote_state.keys()):
            if filename not in remote_files:
                if self.remote_state[filename].exists:
                    self.remote_state[filename] = FileState(
                        mtime=0,
                        size=0,
                        exists=False,
                        last_checked=current_time
                    )
                else:
                    # If already marked as non-existent and not checked for a long time, remove state
                    if current_time - self.remote_state[filename].last_checked > 3600:
                        del self.remote_state[filename]

    def _get_local_file_list(self) -> Dict[str, Dict]:
        """Get list of local directory files"""
        files = {}
        try:
            for item in os.listdir(self.config.local_dir):
                item_path = os.path.join(self.config.local_dir, item)
                if self.is_excluded(item_path):
                    continue
```

```
                try:
                    stat = os.stat(item_path)
                    files[item] = {
                        'size': stat.st_size,
                        'mtime': stat.st_mtime,
                        'is_dir': os.path.isdir(item_path)
                    }
                except Exception as e:
                    self.logger.warning(f"Failed to retrieve file information: {item_path}, Error: {e}")
            return files
        except Exception as e:
            self.logger.error(f"Failed to get local file list: {e}")
            return {}

    def _start_local_watcher(self):
        """Start local file monitoring"""
        event_handler = LocalSyncHandler(self)
        self.observer = Observer()
        self.observer.schedule(event_handler, self.config.local_dir, recursive=True)
        self.observer.start()
        self.logger.info(f"Started monitoring local directory: {self.config.local_dir}")

    def _start_remote_watcher(self):
        """Start remote file monitoring thread"""
        self.remote_watcher_thread = threading.Thread(target=self._watch_remote_changes, daemon=True)
        self.remote_watcher_thread.start()
        self.logger.info("Started monitoring remote directory")

    def _watch_remote_changes(self):
        """Monitor remote file system changes"""
        while self.running:
            try:
                # Get current remote file list
                remote_files = self.sftp.get_remote_file_list()
                if not remote_files:
                    time.sleep(self.config.check_interval)
                    continue

                # Compare with previous state
                for filename in set(remote_files.keys()).union(set(self.remote_state.keys())):
                    remote_info = remote_files.get(filename)
```

```
                    remote_state = self.remote_state.get(filename)

                    # Remote file added
                    if remote_info and (not remote_state or not remote_state.exists):
                        self.logger.info(f"Remote addition detected: {filename}")
                        local_path = os.path.join(self.config.local_dir, filename)
                        remote_path = os.path.join(self.config.remote_dir, filename).replace('\\', '/')
                        self.perform_sync(OperationType.DOWNLOAD, local_path, remote_path)

                    # Remote file deleted
                    elif not remote_info and remote_state and remote_state.exists:
                        self.logger.info(f"Remote deletion detected: {filename}")
                        local_path = os.path.join(self.config.local_dir, filename)
                        # First check if local file exists
                        if os.path.exists(local_path):
                            self.perform_sync(OperationType.DELETE_LOCAL, local_path, None)
                        else:
                            # File already deleted, directly update state
                            rel_path = os.path.relpath(local_path, self.config.local_dir)
                            self.local_state[rel_path] = FileState(
                                mtime=0,
                                size=0,
                                exists=False,
                                last_checked=time.time()
                            )
                            self.logger.info(f"Local file already deleted, updating state directly: {local_path}")

                # Update remote state
                self._update_file_states({}, remote_files)

            except Exception as e:
                self.logger.error(f"Error monitoring remote changes: {e}")

            time.sleep(self.config.check_interval)


# Main program
def main():
    config = Config()

    # Validate configuration
```

```
    if not os.path.exists(config.local_dir):
        print(f"Error: Local directory does not exist - {config.local_dir}")
        return 1

    sync_manager = SyncManager(config)

    try:
        sync_manager.start()
        print("Press Ctrl+C to stop service...")
        while True:
            time.sleep(1)
    except KeyboardInterrupt:
        pass
    finally:
        sync_manager.stop()
        return 0


if __name__ == "__main__":
    main()
```

3.2 Autonomous high-throughput AFM single-cell mechanical analysis on CTC cells

3.2.1 Instruction documentation

(1) Training datasets

Step 1: Create your dataset using LabelImg. Place the original images in VOCdevkit/VOC2007/JPEGImages and the corresponding annotation files in VOCdevkit/VOC2007/Annotations. Run voc_annotation.py to split the dataset into training and test sets. Finally, execute train_yolov7.py to train the YOLOv7 model.

Step 2: Execute crop.py and use the Batch Crop Images function in yolo.py to crop all detected cells from a folder using the trained YOLOv7 model.

Step 3: Label the cropped cells with LabelMe and place the results in the datasets/before folder. Run json_to_dataset.py to convert JSON annotations into PNG format. Store the original images in datasets/JPEGImages and the labels in datasets/SegmentationClass. Transfer these to VOCdevkit_unet/VOC2007, then run voc_annotation_unet.py to partition the training and validation sets. Train the U-Net model with train_unet.py.

(2) AFM automatic detection

Run predict.py.

(3) Additional Notes

1.The program runs in Visual Studio Code.

2.Trained models are saved in the logs/ directory. Ensure you update the model paths in configuration files during testing

3.The integration of U-Net into YOLOv7 is implemented in yolo.py.

4.The combination of template matching and network predictions is handled in yolo.py.

5.Connect the local computer to the JPK microscope control computer via an Ethernet switch, ensuring both devices are on the same local area network (LAN).

6.Copy the code from auto_jpk.txt into JPK NanoWizard software and execute it.

References:

https://github.com/bubbliiiing/yolov7-pytorch

https://github.com/bubbliiiing/yolov7-tiny-pytorch

https://github.com/bubbliiiing/unet-pytorch/tree/bilibili

https://github.com/WongKinYiu/yolov7

3.2.2 Main codes

```
import cv2
import numpy as np
from PIL import Image
import pandas as pd
from yolo import YOLO
import csv
import math
import time
import os
import subprocess
import logging
import shutil
from typing import List, Tuple, Optional, Dict

#--------------------------------------#
#             Synchronization
#--------------------------------------#
# Configure logging
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    filename='sync.log'
)

def generate_script(session_name: str, host: str, username: str, password: str,
                                local_path: str, remote_path: str) -> str:
    """Generate WinSCP script"""
    # Verify local path existence
    if not os.path.exists(local_path):
        raise FileNotFoundError(f"Local path does not exist: {local_path}")
```

```
        # Proper way to disable host key verification
        script = f"""# Sync script - accept any host key
open sftp://{username}:{password}@{host}/ -hostkey=*
option batch on
option confirm off
lcd {local_path}
cd {remote_path}
synchronize both -delete -criteria=time -mirror
close
exit
"""
        script_path = f"{session_name}.txt"

        with open(script_path, "w", encoding="utf-8") as f:
            f.write(script)

        logging.info(f"Script saved to: {os.path.abspath(script_path)}")
        return script_path

def run_winscp(script_path: str, winscp_path: str = r'C:/Program Files (x86)/WinSCP/WinSCP.exe') -> str:
        """Execute WinSCP script"""
        try:
            # Verify script file existence
            if not os.path.exists(script_path):
                raise FileNotFoundError(f"Script file does not exist: {script_path}")

            # Check WinSCP executable existence
            if not os.path.exists(winscp_path):
                raise FileNotFoundError(f"WinSCP executable does not exist: {winscp_path}")

            logging.info("Starting synchronization task")

            # Build and execute command
            command = [
                winscp_path,
                '/script=' + script_path,
                '/log=sync.log',
                '/loglevel=2',
                '/noverifycert'    # Alternative: disable certificate verification
            ]
            logging.info(f"Executing command: {' '.join(command)}")
```

```
        # Use communicate with timeout
        process = subprocess.Popen(
            command,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            text=True
        )

        try:
            stdout, stderr = process.communicate(timeout=300)    # 5-minute timeout
        except subprocess.TimeoutExpired:
            process.kill()
            stdout, stderr = process.communicate()
            raise TimeoutError("Synchronization operation timed out")

        return_code = process.returncode

        if return_code != 0:
            raise subprocess.CalledProcessError(return_code, command, stdout, stderr)

        # Log WinSCP output
        if stdout:
            logging.info(f"WinSCP standard output:\n{stdout}")

        logging.info("Synchronization task completed")
        return stdout

    except subprocess.CalledProcessError as e:
        logging.error(f"Error during synchronization (return code: {e.returncode}):")
        if e.stdout:
            logging.error(f"Standard output:\n{e.stdout}")
        if e.stderr:
            logging.error(f"Error output:\n{e.stderr}")

            # Extract potential WinSCP error messages
            if e.stderr:
                for line in e.stderr.splitlines():
                    if "Error" in line or "Authentication failed" in line:
                        logging.error(f"Critical error: {line}")
```

```
            raise
        except Exception as e:
            logging.error(f"Unexpected error executing WinSCP: {str(e)}")
            raise

def cleanup(script_path: str) -> None:
    """Clean up temporary files"""
    try:
        if os.path.exists(script_path):
            os.remove(script_path)
            logging.info(f"Temporary script deleted: {script_path}")
    except Exception as e:
        logging.error(f"Error cleaning up temporary file: {str(e)}")

def synchronize() -> None:
    """Perform file synchronization"""
    # Configuration
    config = {
        "session_name": "UbuntuSync",
        "host": "10.254.254.1",
        "username": "jpkuser",
        "password": "jpkjpk",
        "local_path": r"C:/Users/qixia/Desktop/RUNJPK",
        "remote_path": r"/home/jpkuser/Desktop/RUNJPK"
    }

    try:
        script_path = generate_script(**config)
    except Exception as e:
        print(f"Failed to generate script: {str(e)}")
        return

    try:
        # Execute synchronization
        output = run_winscp(script_path)
        print("Synchronization successful!")
        if output:
            print(output)
    except Exception as e:
        print(f"Synchronization failed: {str(e)}")
        print(f"Detailed information in log file: {os.path.abspath('sync.log')}")
```

```
        finally:
            cleanup(script_path)    # Clean up temp file regardless of success/failure

#-------------------------------------#
#             Template Matching
#-------------------------------------#
def template_matching_and_save_center(img_path: str) -> Image.Image:
        """Perform template matching and save center coordinates"""
        template_image_path = "template/tp15.jpg"
        csv_path = 'center_coordinates.csv'

        # Read source image
        src_image = Image.open(img_path)
        src_np = np.array(src_image)

        # Read template image
        template_image = Image.open(template_image_path)
        template_np = np.array(template_image)

        match_method = 5    # CV_TM_CCOEFF_NORMED
        result_rows = src_np.shape[0] - template_np.shape[0] + 1
        result_cols = src_np.shape[1] - template_np.shape[1] + 1

        # Perform matching on grayscale
        gray_src = src_np[:, :, 0]
        gray_template = template_np[:, :, 0]
        result_gray = cv2.matchTemplate(gray_src, gray_template, match_method)
        cv2.normalize(result_gray, result_gray, 0, 1, cv2.NORM_MINMAX)

        # Find best match
        min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result_gray)
        match_loc = max_loc    # Use max for TM_CCOEFF_NORMED

        # Calculate center
        center_x = match_loc[0] + template_np.shape[1] // 2
        center_y = match_loc[1] + template_np.shape[0] // 2

        # Save to CSV
        data = {'x': [center_x], 'y': [center_y]}
        df = pd.DataFrame(data)
        df.to_csv(csv_path, index=False)
```

```
    return src_image

#-------------------------------------#
#           Find Densest Point
#-------------------------------------#
def find_nearest_center(input_file: str) -> Optional[Tuple[float, float]]:
    """Find the center of the densest point cluster"""
    points = []
    with open(input_file, 'r', encoding='utf-8', newline='') as f:
        reader = csv.reader(f)
        for row in reader:
            try:
                x = float(row[0])
                y = float(row[1])
                if abs(x) > 4.9 and abs(y) > 4.9 and y < 0:
                    points.append((x, y))
            except (IndexError, ValueError):
                continue

    # Check data sufficiency
    if len(points) < 1:
        print("Error: At least one valid coordinate point is required")
        return None

    # Find densest point
    max_point_count = 0
    best_point = None
    for center_point in points:
        point_count = 0
        for point in points:
            if (center_point[0] - 3 <= point[0] <= center_point[0] + 3) and (
                    center_point[1] - 6 <= point[1] <= center_point[1] + 6):
                point_count += 1

        if point_count > max_point_count:
            max_point_count = point_count
            best_point = center_point

    return (
        round(best_point[0], 2),
```

```
            round(best_point[1], 2)
        ) if best_point else None

#-------------------------------------#
#           Draw Points on Image
#-------------------------------------#
def draw_points_on_image(image: np.ndarray, csv_file_paths: List[str],
                         point_color: Tuple[int, int, int] = (0, 125, 0),
                         point_size: int = 5) -> np.ndarray:
    """Draw points from CSV files on an image"""
    for csv_file_path in csv_file_paths:
        try:
            with open(csv_file_path, 'r') as csvfile:
                reader = csv.reader(csvfile)
                for row in reader:
                    if len(row) == 2:
                        try:
                            x = int(float(row[0]))
                            y = int(float(row[1]))
                            # Check if coordinates are within image bounds
                            if 0 <= x < image.shape[1] and 0 <= y < image.shape[0]:
                                cv2.circle(image, (x, y), point_size, point_color, -1)
                        except ValueError:
                            print(f"Invalid coordinate value: {row}")
        except FileNotFoundError:
            print(f"Error: File not found {csv_file_path}")
        except Exception as e:
            print(f"Unknown error: {e}")
    return image

#-------------------------------------#
#           Convert TIF to JPG
#-------------------------------------#
def tif_to_jpg(input_path: str, output_path: str) -> bool:
    """Convert TIF image to JPG"""
    try:
        # Open TIF image
        image = Image.open(input_path)
        # Convert to RGB mode
        rgb_image = image.convert('RGB')
        # Save as JPG
```

```
            rgb_image.save(output_path, 'JPEG')
            print(f"Successfully converted {input_path} to {output_path}")
            return True
        except Exception as e:
            print(f"Error during conversion: {e}")
            return False

#-------------------------------------#
#           Utility Functions
#-------------------------------------#
def find_nearest(current: Tuple[float, float], points: List[Tuple[float, float]]) -> Optional[Tuple[float, float]]:
    """Find the nearest point to the current point"""
    if not points:
        return None
    return min(points, key=lambda p: math.hypot(p[0]-current[0], p[1]-current[1]))

def clear_folder(folder_path: str) -> None:
    """Clear all files and subfolders in a folder"""
    if not os.path.exists(folder_path):
        print(f"Folder does not exist: {folder_path}")
        return

    # Iterate through all items
    for item in os.listdir(folder_path):
        item_path = os.path.join(folder_path, item)
        try:
            if os.path.isfile(item_path):
                os.remove(item_path)
                # print(f"File deleted: {item_path}")    # Commented for performance
            elif os.path.isdir(item_path):
                shutil.rmtree(item_path)
                # print(f"Folder deleted: {item_path}")    # Commented for performance
        except Exception as e:
            print(f"Failed to delete {item_path}: {e}")

def delete_file(file_path: str) -> None:
    """Delete a file with error handling"""
    try:
        if os.path.exists(file_path):
            os.remove(file_path)
            # print(f"File deleted: {file_path}")    # Commented for performance
```

```
            else:
                # print(f"File not found: {file_path}")    # Commented for performance
                pass
        except PermissionError:
            print(f"Permission error: Cannot delete {file_path}, possibly in use by another program")
        except Exception as e:
            print(f"Unknown error: {e}, cannot delete {file_path}")

#-------------------------------------#
#           Main Function
#-------------------------------------#
if __name__ == "__main__":
    yolo = YOLO()
    crop = True
    count = False

    # File paths
    input_file = 'coordinates.csv'
    output_file = 'C:/Users/qixia/Desktop/RUNJPK/output.csv'
    output_file2 = 'C:/Users/qixia/Desktop/RUNJPK/output2.csv'
    test_file = 'C:/Users/qixia/Desktop/RUNJPK/test.csv'
    image_extensions = ('.jpg', '.jpeg', '.png', '.gif', '.tif')
    usb_drive_path = 'C:/Users/qixia/Desktop/RUNJPK'
    input_tif_file = 'C:/Users/qixia/Desktop/RUNJPK/1.tif'
    test_tif = 'C:/Users/qixia/Desktop/RUNJPK/2.tif'
    output_jpg_file = 'C:/Users/qixia/Desktop/RUNJPK/1.jpg'
    csv_file_paths = [output_file, output_file2]

    while True:
        clear_folder(usb_drive_path)
        print(f"Folder cleared: {usb_drive_path}")
        synchronize()
        print(f"Start detection: {usb_drive_path}")

        # Wait for test image
        while not os.path.exists(test_tif):
            print("Test image not detected, waiting...")
            time.sleep(5)    # Check every 5 seconds
            delete_file(output_file)
            delete_file(output_file2)
            synchronize()
```

```
        print(f"Start detection: {usb_drive_path}")

try:
        delete_file(output_file)
        delete_file(output_file2)

        # Verify image integrity
        img = Image.open(input_tif_file)
        img.verify()
        img.close()
        is_corrupted = False
except (IOError, SyntaxError):
        delete_file(output_file)
        delete_file(output_file2)
        is_corrupted = True

if is_corrupted:
        print("Detected corrupted image.")
        with open(output_file, 'w') as f:
                f.write("0,0")
        with open(output_file2, 'w') as f:
                f.write("0,0")
        with open(test_file, 'w') as f:
                pass    # Empty file
        print("Test CSV file generated!")
else:
        # Initialize coordinates file
        with open("coordinates.csv", "w") as f:
                pass    # Empty file

        # Convert TIF to JPG
        if tif_to_jpg(input_tif_file, output_jpg_file):
                # Perform template matching
                result_image = template_matching_and_save_center(output_jpg_file)

                # Perform object detection
                r_image = yolo.detect_image(result_image, crop=crop, count=count)

                ############ Write coordinates with |x|<5 and |y|<5 to output.csv ################
                try:
                        # Read all coordinates
```

```
coords = []
with open(input_file, 'r', encoding='utf-8', newline='') as infile:
    reader = csv.reader(infile)
    for row in reader:
        try:
            x = float(row[0])
            y = float(row[1])
            if abs(x) < 5 and abs(y) < 5:
                coords.append((x, y))
        except (IndexError, ValueError):
            continue

# Handle empty coordinates
if not coords:
    x, y = 0.0, 0.0
    coords.append((x, y))
    print("No valid coordinates found!")

# Greedy algorithm for nearest neighbor ordering
path = []
unvisited = set(coords)

# Start from point nearest to origin
start = min(coords, key=lambda p: math.hypot(p[0], p[1]))
path.append(start)
unvisited.remove(start)

# Find nearest points sequentially
current = start
while unvisited:
    next_point = find_nearest(current, unvisited)
    path.append(next_point)
    unvisited.remove(next_point)
    current = next_point

# Write ordered coordinates
with open(output_file, 'w', encoding='utf-8', newline='') as outfile:
    writer = csv.writer(outfile)
    for x, y in path:
        writer.writerow([x, y])
        # print(f"Target point ({x}, {y}) saved to {output_file}")  # Commented for performance
```

```
    except FileNotFoundError:
        print(f"Error: File not found {input_file}.")
    except Exception as e:
        print(f"Unknown error: {e}")

###################### Find nearest center point ###################################
# Find nearest center point
result = find_nearest_center(input_file)
# Write to CSV file
if result:
    with open(output_file2, 'w', newline='', encoding='utf-8') as f:
        writer = csv.writer(f)
        writer.writerow(result)
    print(f"Nearest center coordinates {result} saved to {output_file2}")
else:
    with open(output_file2, 'w', newline='', encoding='utf-8') as f:
        writer = csv.writer(f)
        writer.writerow([0, 0])
    print(f"Default coordinates (0, 0) saved to {output_file2}")

###################### Process and save result image ###################################
if r_image and len(r_image) > 0:
    image_obj = r_image[0]    # Assuming r_image is a tuple containing the image
    # Convert PIL image to numpy array
    image_np = np.array(image_obj)
    # Convert RGB to BGR for OpenCV
    if len(image_np.shape) == 3 and image_np.shape[2] == 3:
        image_np = cv2.cvtColor(image_np, cv2.COLOR_RGB2BGR)

    # Draw points on image
    try:
        result_image = draw_points_on_image(image_np, csv_file_paths)
        # Save image as output.jpg
        cv2.imwrite('output.jpg', result_image)
        print("Image successfully saved as output.jpg")
    except Exception as e:
        print(f"Error saving image: {e}")
        # Fallback to saving original image
        cv2.imwrite('output.jpg', image_np)
else:
```

```
            print("No detection result image available")
            # Create a blank image as fallback
            blank_image = np.zeros((600, 800, 3), np.uint8)
            cv2.imwrite('output.jpg', blank_image)

        # Generate empty test CSV file
        with open(test_file, 'w') as f:
            pass    # Empty file
        print("Test CSV file generated!")

        # Clean up image files
        for root, dirs, files in os.walk(usb_drive_path):
            for file in files:
                if file.lower().endswith(image_extensions):
                    file_path = os.path.join(root, file)
                    try:
                        os.remove(file_path)
                        # print(f"File deleted: {file_path}")    # Commented for performance
                    except Exception as e:
                        print(f"Failed to delete {file_path}: {e}")

        # Perform synchronization
        synchronize()
        wait_time = 1
        print(f"Synchronization completed, waiting {wait_time} second")
        time.sleep(wait_time)
```

3.2.3 AFM script

```
import time
import os
import csv
# Check the SPM version
checkVersion('SPM', 7, 0, 178)

# The path to the output table file
file_path = '/home/jpkuser/Desktop/RUNJPK/output.csv'
# The path to the output table file
file_path2 = '/home/jpkuser/Desktop/RUNJPK//output2.csv'
test_path = '/home/jpkuser/Desktop/RUNJPK//test.csv'
# File name
filename = '/home/jpkuser/Desktop/RUNJPK/1'
testname = '/home/jpkuser/Desktop/RUNJPK/2'
```

```
image1 = '/home/jpkuser/Desktop/RUNJPK/1.tif'
image2 = '/home/jpkuser/Desktop/RUNJPK/2.tif'
# Wait for the file corresponding to the file path to appear
def wait_for_file_path(file_path):
    while not os.path.exists(file_path):
        pass


# Main function
while True:
    # Securely delete files
    for path in [file_path, file_path2,test_path,image1,image2]:
        try:
            os.remove(path)
        except:
            pass
    print("Remote all file")
    Snapshooter.saveOpticalSnapshot(filename)                # Save the image
    time.sleep(3.0)
    Snapshooter.saveOpticalSnapshot(testname)                # Save the image
    print("Save the image")

    #-------------Initialization-------------#

    MotorizedStage.disengage()                                  # Disable platform movement
    ForceSpectroscopy.clearPositions()                       # Clear the coordinates and read new ones
    ForceSpectroscopy.addPosition(0, 0)                      # Add the initial position to the software table and set it as
the origin with index 0
    ForceSpectroscopy.moveToForcePositionIndex(0)          # Move the probe back to the initial position

    coordinate_count = 0                                        # Variable initialization

    #-------------Small area movement-------------#
    wait_for_file_path(test_path)                            # Wait for the file to appear
    print("The path has been successfully detected")
    # Securely delete files
    for path in [test_path,image1,image2]:
        try:
            os.remove(path)
        except:
            pass
```

```
print("remote the image1")
with open(file_path, mode='r') as file:                    # Read the coordinates of detectable points
    reader = csv.reader(file)
    for row in reader:
        x = float(row[0])*1e-5
        y = float(row[1])*1e-5
        ForceSpectroscopy.addPosition(x, y)                # Add coordinate points
        coordinate_count += 1          # Count
i=0
for j in xrange(coordinate_count):
    i+=1
    ForceSpectroscopy.moveToForcePositionIndex(i)      # Move to the specified force position index
    Scanner.approach()                                     # Lower the probe
    ForceSpectroscopy.startScanning(5)                     # Start force spectroscopy scanning, scan 5 times
    Scanner.retractPiezo()                                 # Raise the probe
    Scanner.moveMotorsUp(2e-5)                             # Raise the probe height
    time.sleep(1.0)

ForceSpectroscopy.addPosition(0, 0)                        # The probe must return to zero

#-----------Move the probe in a wide area----------#
for path in [test_path,image1,image2]:
    try:
        os.remove(path)
    except:
        pass
print("remote the image2")
MotorizedStage.engage()                                    # Enable platform movement
#--Read the large-scale movement coordinates and add them to the JPK system--#
with open(file_path2, mode='r') as file:                   # Read the coordinates of detectable points
    reader = csv.reader(file)
    for row in reader:
        next_x = float(row[0])*1e-5
        next_y = float(row[1])*1e-5
        MotorizedStage.moveToRelativePosition(next_x,next_y)    # Move to the next position on a large scale

MotorizedStage.disengage()                                 # Disable the moving platform
```