TDT4127 Programming and Numerics
Week 46

Repetition and exam preparation

Next week

- Questions about the exam:
  - Friday November 23, 16:15-17:00
  - Bring your questions, Guttorm and I will bring our answers
  - Afterward, 17:00 – 18:00: Final exam prep

Today

- Finalize adaptive Simpson's method
  - Going through implementation
- Repetition
  - Summarize what we've learned
  - Go through exercises
- Exam preparation
  - What do we need to know?
  - What can be expected in the exam?
  - How do we prepare?

Implementing Adaptive Simpson's rule

$S(a, b)$ denotes Simpson's on the integral from a to b.

To approximate the integral over $[a, b]$ with error $< \epsilon$:

1. Compute $S(a, b)$.
2. Compute $S(a, c)$ and $S(c, b)$.
3. Estimate the error in $S(a, c) + S(c, b)$:

   if $\left|S(a,b) - \left(S(a,c) + S(c,b)\right)\right| < 15 * \epsilon$:

   return $\frac{16}{15}\left(S(a,c) + S(c,b)\right) - \frac{1}{15}S(a,b)$

   else:

   estimate the integrals over $[a, c]$ and $[c, b]$ with error less than $\epsilon/2$
   return the two estimates added together

NTNU

Repetition

Week 35/36: Number representation

- Computers mainly use two storage formats for numbers: Integers and floating point numbers (floats)
- **Integers: Precise** representations of whole numbers
  - Used for *counting*, *numbering* etc.
  - **Format**: Binary numbers. 8-bit example:

    10010101 = 1*128 + 0* 64 + 0*32 + 1*16 + 0*8 + 1*4 + 0*2 + 1*1 = 149
  - More bits ⇔ can represent larger numbers
  - First bit may represent the sign (0 means negative, 1 positive)

Week 35/36: Number representation

- **Floating point numbers:** **Imprecise** versions of real numbers
  - Used in *calculations* requiring *decimal points*
  - **Format:** Scientific notation in base 2 (totallsystemet)

    $$a = (-1)^{sg} \times 2^{e-b} \times 1.s_1 s_2 s_3 \dots s_K$$

    - sg: sign, e: exponent, b: bias, $1.s_1 s_2 s_3 \dots s_K$: significand/mantissa
  - Due to imprecision, be careful with floating point operations:
    - $a \pm b$ is problematic if a and b are **very different in size**
    - $a \times b$ and a/b are **safe**
    - $a == b$ is very unsafe and should be avoided (check $|a - b| < \epsilon$ instead)

Week 36/38/39: Equation solvers

- Solving $f(x) = g(x) \Leftrightarrow$ solving $h(x) = f(x) - g(x) = 0$
  - Therefore the algorithms are based on solving $h(x) = 0$.
- Three methods: bisection, secant and Newton's
  - Newton uses **derivative.** Secant and bisection: **derivative free**
  - Newton is **faster** than secant which is **faster** than bisection
  - Bisection has **less rigid** restrictions than secant which has **less rigid** restrictions than Newton

Property type	Newton's method	Secant method	Bisection method
Continuity	f''	f'	f
Nonzero	$f''(z) \neq 0, f'(x) \neq 0$	$f'(z) \neq 0$	None
Extra bounds	$\frac{\lvert f''(x)\rvert}{\lvert f'(y)\rvert} \leq A$	None	None
Starting point	Close enough	Close enough	$[a, b]$encloses z

Algorithm: Bisection method

- **Type:** Equation solver. Finds zeroes: $f(x) = 0$
- **Initialization:** $[a, b]$ such that $f(a)$ and $f(b)$ have different signs ($f(a)f(b) < 0$), a minimum width ϵ.
- **Mathematically:** Halve the interval, but ensure $f(a)f(b) < 0$
- **Pseudoalgorithm:**

```
while abs(a-b) > epsilon:
    c = (a+b)/2
    if f(a) and f(c) have the same sign:
        a = c
    else:
        b = c
    if f(c) is 0:
        return c
return c
```

Algorithm: Newton's method

- **Type:** Equation solver. Finds zeroes: $f(x) = 0$
- **Initialization:** Starting value x_0, tolerances ϵ, δ.
- **Mathematically:** $x_{k+1} = x_k - f(x_k)/f'(x_k)$
- **Algorithm:**

```
k = 0
diff = delta + 1
while f(x_k) > epsilon and diff > delta
    x_{k+1} = x_k − f(x_k)/f'(x_k)
    diff = x_{k+1} - x_k
    k = k+1
return x_{k+1}
```

- **Note:** Requires the derivative $f'(x)$

Algorithm: Secant method

- **Type:** Equation solver. Finds zeroes: $f(x) = 0$
- **Initialization:** Starting values x_0 and x_1, tolerances ϵ, δ.
- **Mathematically:** $x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$
- **Algorithm:**

```
k = 1
while f(x_k) > epsilon and abs(x_k − x_{k−1}) > delta
    x_{k+1} = x_k − f(x_k)(x_k - x_{k−1})/(f(x_k) − f(x_{k−1}))
    k = k+1
return x_{k+1}
```

- **Note:** Can be seen as a derivative-free version of Newton's

Week 37/45: Numerical integration

- **Task:** Compute a definite integral $\int_a^b f(x)\mathrm{d}x$
- **Three methods**: Midpoint, trapezoidal, Simpson's rule
  - Based on: constant, linear and quadratic approximations of f.
  - Simpson's rule is a bit more work but also more accurate
- **Composite** methods: Split $[a, b]$ into N parts, integrate each part separately, add together.
- **Error analysis**, $M_2 = \max_{a \le y \le b} f''(y)$, $M_4 = \max_{a \le y \le b} f''''(y)$:

$$E_{MP} \le \frac{(b-a)^3}{24N^2} M_2, \qquad E_{TR} \le \frac{(b-a)^3}{12N^2} M_2, \qquad E_{TR} \le \frac{(b-a)^5}{2880N^4} M_4$$

- **Adaptive Simpson's rule** uses error analysis/recursion
  - More efficient than composite methods, guarantees error

Algorithm: Composite Midpoint rule

- **Type:** Integral computing. Finds $\int_a^b f(x)\mathrm{d}x$
- **Initialization:** $[a, b]$, number of intervals N
- **Mathematically:**

$$\int_a^b f(x)\mathrm{d}x \approx h \sum_{k=0}^{N-1} f\left(x_k + \frac{h}{2}\right), \qquad h = \frac{a-b}{N}, \qquad x_k = a + kh$$

- **Algorithm:**

```
h = (b-a)/N
totalSum = 0
for k in range(0,N)
    x_k = a + k*h
    totalSum += f(x_k + h/2)
totalSum = h*totalSum
return totalSum
```

Algorithm: Composite Trapezoidal rule

- **Type:** Integral computing. Finds $\int_a^b f(x)\mathrm{d}x$
- **Initialization:** $[a, b]$, number of intervals N
- **Mathematically:**

$$\int_a^b f(x)\mathrm{d}x \approx \frac{h}{2}\left(f(x_0) + 2\sum_{k=1}^{N-1} f(x_k) + f(x_N)\right), \qquad h = \frac{a-b}{N}, x_k = a + kh$$

- **Algorithm:**

```
h = (b-a)/N
totalSum = f(a)
for k in range(1,N)
    x_k = a + k*h
    totalSum += 2*f(x_k)
totalSum += f(b)
totalSum = h/2*totalSum
return totalSum
```

Algorithm: Composite Simpson's rule

- **Type:** Integral computing. Finds $\int_a^b f(x)\mathrm{d}x$
- **Initialization:** $[a, b]$, number of intervals N
- **Mathematically:**

$$\int_a^b f(x)\mathrm{d}x \approx \frac{h}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 2f(x_{2N-2}) + 4f(x_{2N-1}) + f(x_{2N})\right)$$

$$h = \frac{a-b}{2N}, \qquad x_k = a + kh$$

- **Algorithm:**

```
h = (b-a)/(2*N)
totalSum = f(a)
for k in range(1,2N)
    x_k = a + k*h
    if k % 2 is 1:  # Odd index
        totalSum += 4*f(x_k)
    else:           # Even index
        totalsum += 2*f(x_k)
totalSum += f(b)
totalSum = h/3*totalSum
return totalSum
```

Algorithm: Adaptive Simpson's rule

- **Type:** Integral computing. Finds $\int_a^b f(x)\mathrm{d}x$
- **Initialization:** $[a, b]$, error tolerance ϵ
- **Algorithm:**

```
def ad_Simpson(f,a,b,eps)
    whole = Simpson(f,a,b)
    c = (a+b)/2
    left = Simpson(f,a,c)
    right = Simpson(f,b,c)
    if abs(whole – (left + right)) < 15*eps:     # Error OK
        return 16/15*(left + right) – 1/15*whole # Extrapolation
    else:   # Error not OK, split interval in two
        return ad_Simpson(f,a,c,eps/2) + ad_Simpson(f,c,b,eps/2)
```

Week 40/41: Gaussian elimination

- **Task:** Solve a matrix-vector system $A\boldsymbol{x} = \boldsymbol{b}$
- **The method:** Gaussian elimination + back substitution
- GE is a ***direct*** solver: Running the algorithm **gives the answer**, no iterations or error estimates
- Roundoff errors are minimized by **partial pivoting**
  - Swap rows such that the pivot element is maximal in its column
- After Gaussian elimination, use ***back substitution*** to find the answer
- Can be implemented ***in-place***; don't need to create new matrices, saves space

Algorithm: Gaussian elimination with partial pivoting

- **Type:** Linear equation solver. Solves: $A\boldsymbol{x} = \boldsymbol{b}$
- **Initialization:** $N \times (N+1)$ augmented matrix M
- **Pseudoalgorithm:**

```
row = 0, col = 0
while (row < N-1 and col < N):
    ind_row_max = get_max(M,row,col) # Maximum in col
    if w_max][col] is 0: # Pivot element is 0
        col += 1  # No nonzero element in pivot column
    else:
        swap(M[row_ind],M[max_row_ind]) # Swap rows
        row_reduce(M,row,col) # Zero out rows below
        row += 1, col += 1
x = back_substitute(M) # Back substitution
```

Week 42: Newton's method in n-D

- **Task**: Solve $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$
- Very similar to the 1-D version, uses the **Jacobian matrix**

$$J_{\boldsymbol{f}}(\boldsymbol{y}) = \begin{bmatrix} \frac{\partial f_0}{\partial x_0}(\boldsymbol{y}) & \frac{\partial f_0}{\partial x_1}(\boldsymbol{y}) & \cdots & \frac{\partial f_0}{\partial x_n}(\boldsymbol{y}) \\ \frac{\partial f_1}{\partial x_0}(\boldsymbol{y}) & \frac{\partial f_1}{\partial x_1}(\boldsymbol{y}) & \cdots & \frac{\partial f_1}{\partial x_n}(\boldsymbol{y}) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_0}(\boldsymbol{y}) & \frac{\partial f_n}{\partial x_1}(\boldsymbol{y}) & \cdots & \frac{\partial f_n}{\partial x_n}(\boldsymbol{y}) \end{bmatrix}$$

1. Solve the linear system $J_{\boldsymbol{f}}\left(\boldsymbol{x}^k\right)\boldsymbol{z} = -\boldsymbol{f}\left(\boldsymbol{x}^k\right)$
2. Compute $\boldsymbol{x}^{\boldsymbol{k}+\mathbf{1}} = \boldsymbol{x}^{\boldsymbol{k}} + \boldsymbol{z}$

- Stopping conditions must take all dimensions into account
  - Example: $\left|f_0\left(\boldsymbol{x}^{\boldsymbol{k}}\right)\right| < \epsilon$ **and** $\left|f_1\left(\boldsymbol{x}^{\boldsymbol{k}}\right)\right| < \epsilon$ … **and** $\left|f_n\left(\boldsymbol{x}^{\boldsymbol{k}}\right)\right| < \epsilon$,
    and/or: $\left|x_0^k - x_0^{k-1}\right| < \delta$ and … and $\left|x_n^k - x_n^{k-1}\right| < \delta$.

Algorithm: Newton's method in n-D

- **Type:** Equation solver. Finds zeroes: $\boldsymbol{f}(\boldsymbol{x}) = 0$
- **Initialization:** $\boldsymbol{x}^{\mathbf{0}}$, tolerances ϵ, δ.
- **Mathematically:**
  - Solve the linear system $J_{\boldsymbol{f}}(\boldsymbol{x}^k)\boldsymbol{z} = -\boldsymbol{f}(\boldsymbol{x}^k)$
  - Compute $\boldsymbol{x}^{\boldsymbol{k}+\mathbf{1}} = \boldsymbol{x}^{\boldsymbol{k}} + \boldsymbol{z}$
- **Pseudoalgorithm:**

```
k = 0
while <stopping conditions are not satisfied>
    compute J_f(x^k), f(x^k)
    solve the linear system J_f(x^k) z = -f(x^k)
    x^(k+1) = x^k + z
    k += 1
return x^k
```

Week 43/44: Methods for solving ODEs

- **Task**: Solve the ODE $\dot{x}(t) = f(x, t)$; solution is $x(t)$
- **Numerically**: find a series $\{x^k\}_{k=0}^{N}$, $x^k \approx x(kh)$
- Formulation of methods is *the same for 1-D and n-D*
- Methods can be **explicit** or **implicit**
  - **Explicit:** $\boldsymbol{x^k}$ can be computed directly (explicit Euler, Heun's)
  - **Implicit:** $\boldsymbol{x^k}$ is computed by solving an equation (implicit Euler)
- Methods can have several ***stages***
  - Combine several estimates of the slope to get a better fit.
    - Heun's method is a 2-stage method
- **Stability**
  - A method is unstable if $x^k \to \infty$ as $\mathrm{k} \to \infty$ when applied to the test equation
    $$f(x, t) = -\lambda x, \qquad \lambda \geq 0$$
  - **Implicit** methods are often **more stable but slower** than explicit methods
- **Convergence order**
  - A method is of order p if $|x^k - x(kh)| < C_k h^p$
  - An order p method improves its answer by a factor 2^p when $h \to h/2$
  - Explicit/Implicit Euler are order 1, Heun's method order 2

Algorithm: Explicit Euler

- **Type:** ODE solvers. $\dot{x}(t) = f(x, t),\ x(0) = x^0$
- **Initialization:** x^0, T, N
- **Mathematically:** $x^{j+1} = x^j + hf(x^j, t_j)$
- **Pseudoalgorithm:**

```
x_list = [x⁰]
x = x⁰
h = T/N
for j in range(N)
    x = x + hf(x,jh)
    x_list.append(x)
return x_list
```

Algorithm: Implicit Euler

- **Type:** ODE solvers. $\dot{x}(t) = f(x,t),\ x(0) = x^0$
- **Initialization:** x^0, T, N
- **Mathematically:** $x^{j+1} = x^j + hf\left(x^{j+1}, t_{j+1}\right)$
- **Pseudoalgorithm:**

```
x_list = [x⁰]
x = x⁰
h = T/N
for j in range(N)
    solve the equation y = x + hf(y,(j+1)h)
    x = y
    x_list.append(x)
return x_list
```

Algorithm: Heun's method

- **Type:** ODE solvers. $\dot{x}(t) = f(x, t)$, $x(0) = x^0$
- **Initialization:** x^0, T, N
- **Mathematically:** $s^{j+1} = x^j + hf\left(x^j, t_j\right)$

$$x^{j+1} = x^j + \frac{h}{2}\left(f\left(x^j, t_j\right) + f\left(s^{j+1}, t_{j+1}\right)\right)$$

- **Pseudoalgorithm:**

```
x_list = [x^0]
x = x^0
h = T/N
for j in range(N)
    s = x + hf(x,jh)
    x = x + h/2*(f(x,jh) + f(s,(j+1)h))
    x_list.append(x)
return x_list
```

Week 41: Plotting

- Include matplotlib using the command

```
import matplotlib.pyplot as plt
```

- Given lists `x` and `y` of equal length, we plot the points (`x[i],y[i]`) with the command `plt.plot(x,y)`
  - Same as when drawing a graph from hand if you have no idea how it looks: put dots on the coordinates and draw lines between
- To see the figure, use `plt.show()`

```
#Inform about label on the y axis
plt.ylabel('some numbers')
#Axis range: [x_min, x_max, y_min, y_max]
plt.axis([0,4,0,16])
```

Plotting styles

- The default behaviour of `plt.plot()` is to connect the points with lines
- We can change this using additional arguments after the `x/y` coordinates
  - For example, to plot `y` over the `x` points as red **circles**:

    `plt.plot(x,y,'ro')`
  - To plot `y` over the `x` points as green **triangles**:

    `plt.plot(x,y,'g^')`

Plotting several graphs in one figure

- If we want to generate several graphs, plot all of them first using `plt.plot()`, then use `plt.show()`

```
#Import plotting library
import matplotlib.pyplot as plt
x = …
y1 = f(x)
y2 = g(x)
plt.plot(x,y1)
plt.plot(x,y2)
plt.show()
```

Questions?