1 Floating and error

*	-Inf	-0	0	Inf	NaN
-Inf	Inf	NaN	NaN	-Inf	NaN
-0	NaN	0	-0	NaN	NaN
0	NaN	-0	0	NaN	NaN
Inf	-Inf	NaN	NaN	Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN

Inf : > any other FP value & overflow "flag"
NaN : indeterminate/invalid computation, propagates
IEEE754 valid : $x+y=y+x, x+x=2\cdot x, x-x=0, \frac{x}{2}\cdot 2=x, (2x)\cdot\frac{y}{2}=x\cdot y, -x=-1\cdot x$
IEEE754 invalid : $x+2\cdot y=(x+y)+y=x+(y+y), x<x+y\,(x,y>0), \mathrm{NaN}=\mathrm{NaN}, \sqrt{x^2}=|x|, x+y-y=x$
IEEE : guaranties elementary but not distribution
Demoralized numbers : fill gap btw. numbers, very small values at best possible FP discretion instead of flushing to 0, leading 0 as exp., slower

1.1 Error

Absolute (forward) : $y-\hat{y}$
Relative : $\frac{\text{absolute}}{y}$
Backward : $x-f^{-1}(\hat{y})$
Condition number : $\frac{\text{forward}}{\text{backward}}$
Truncation : diff. btw. actual value and value with digits chapped off
Rounding : loss of prec. after ULP (space btw. 2 consec. FP val.)
Cancellation : loss of significance when subtracting 2 close FP numb., tiny residual can move to leading position, happens even when res. is exactly representable
Over/underflow : value too big/small for numb. of bits allocated

2 Linear Systems

$f(\alpha x)=\alpha f(x), \quad \alpha\in\mathbb{R}$
$f(x+y)=f(x)+f(y)$
$(AB)_{ij}=\sum_{k=1}^{n}A_{ik}B_{kj}$

Linear transformations : (combinations of) scale, shear, rotate, flip
Over;Under-determined : $m>n$ (tall), no solution in general;$m<n$ (wide), none or infinite nb. solutions
Gaussian elimination : $O(n^3)$, select pivot in row, scale pivot to 1, sub-mult of row to create 0 in col below
Back-substitution : $O(n^2)$, zero out col entries above

2.1 LU decomposition

Poor accuracy if ill-conditioned

$$\boxed{A}=\boxed{L}\boxed{U}, \quad \boxed{{}_{L}{}^{U}}$$

Store in single matrice (L's diagonal = 1).
$U=E_n\dots E_1A, L=E_1^{-1}\dots E_n^{-1}$
$A\vec{x}=\vec{b}\implies \vec{x}=U^{-1}(L^{-1}\vec{b})$
Partial;full-pivoting : $PA=LU, P_1LUP_2$

Solve : apply P to b n, △ solve using L n^2, △ solve using U n^2

```
lu, piv = scypy.linalg.lu_factor(A)
x=scipy.linalg.lu_solve((lu,piv),b)
```

Cholesky decomposition : $A=LL^T, \begin{pmatrix}\sqrt{A_{11}} & 0\\ \frac{A_{21}}{L_{11}} & \sqrt{A_{22}-L_{21}^2}\end{pmatrix}$
Solve : △ solve with L n^2, △ solve with L^T n^2

3 Conditioning

$\hat{b}=A\hat{x}\approx b$
Residual : $\vec{r}(\hat{x}):=A\hat{x}-\vec{b}$
Norm : $\|\vec{x}\|_p=\sqrt[p]{\sum_{i=1}^{n}|\vec{x}_i|^p}, \|\vec{x}\|_\infty=\max_{i=1}^{n}|\vec{x}_i|, \|\alpha\vec{x}\|=|\alpha|\cdot\|\vec{x}\|, \|\vec{x}+\vec{y}\|\le\|x\|+\|y\|$
Matrix norm : $\|A\|=\max\{\|A\vec{x}\|:\|x\|=1\}, \|AB\|\le\|A\|\|B\|, \quad \|A\vec{x}\|\le\|A\|\|\vec{x}\|$
$A\vec{x}=\vec{b}\implies\|A\|\|x\|\ge\|b\|$

3.1 Condition number

Ill-conditioned : sensitive to input noise, exploding problem
$\mathrm{cond}(A)=\|A\|\cdot\|A^{-1}\|$
`np.linalg.cond(A, p=2)`
$\mathrm{cond}(A)=\infty\iff A$ is singular
$A=I_n\implies\mathrm{cond}(A)=1\impliedby A$ scaled orthogonal matrix ($A=\sigma UV^T$)
$\mathrm{cond}(\alpha A)=\mathrm{cond}(A), \quad \alpha\in\mathbb{R}$
$\mathrm{cond}(AA^T)\approx\mathrm{cond}(A)^2$

Perturbed system : $A(\vec{x}+\Delta\vec{x})=\vec{b}+\Delta\vec{b}$
$\frac{\|\Delta\vec{x}\|}{\|\vec{x}\|}\le\|A^{-1}\|\cdot\left\|\Delta\vec{b}\right\|\cdot\frac{\|A\|}{\|\vec{b}\|}$
$\frac{\|\Delta\vec{x}\|}{\|\vec{x}\|}\lesssim\mathrm{cond}(A)\mathrm{ulp}(1)$
$\mathrm{ulp}(1)\approx 2^{-23};2^{-52}$ (double)

3.2 Least squares

Minimise $\|\vec{r}(\vec{x})\|_2=\left\|A\vec{x}-\vec{b}\right\|_2$
Least squares :
$\arg_{a,b}\min\sum_{i=1}^{n}((ax_i+b)-y_i)^2$
Weighted least squares :
$\arg_{a,b}\min\sum_{i=1}^{n}(w_i(ax_i+b)-y_i)^2$, $w_i=\frac{1}{\sigma_i}$, σ_i std. deviation
Normal equations : $A^TA\vec{x}=A^T\vec{b}$

Regularization : improve the condition number (decreased accuracy, bias, ...)
Tikhonov Regularization :
$$\vec{x}_{\text{sol}}=\arg\min\left[\left\|A\vec{x}-\vec{b}\right\|_2^2+\lambda\|\vec{x}\|_2^2\right]$$
λ : parameter controlling the degree to which we prefer smaller solutions
LU : add λI to A^TA via normal equation
QR : factorize extended $\begin{bmatrix}A\\ \sqrt{\lambda I}\end{bmatrix}\vec{x}=\begin{bmatrix}\vec{b}\\ 0\end{bmatrix}$

3.3 QR factorization

Good accuracy if system ill-conditioned, slower than LU
$\vec{x}^TA^TA\vec{x}=\langle A\vec{x},A\vec{x}\rangle=\|A\vec{x}\|^2$
$\vec{a}\cdot\vec{b}=\vec{a}^T\vec{b}=\|\vec{a}\|\left\|\vec{b}\right\|\cos\theta$
$\vec{a}\perp\vec{b}\implies\vec{a}\cdot\vec{b}=0$

Orthogonal transformations : preserve distances and angles, do not amplify error
Householder reflection : excellent numerical properties, hard to parallelize
find Q such that $Q\vec{x}=(\widetilde{x}_1\quad 0\quad\cdots)^T$, $\widetilde{x}=\vec{x}-2(\vec{v}^T\vec{x})\vec{v}, (\|\vec{v}\|=1)$, $\widetilde{x}=Q\vec{x}, Q=I-2\frac{vv^T}{v^Tv}$

$$\mathbf{x}=\boxed{\mathbf{R}}^{-1}\left(\boxed{\mathbf{Q}^T}\ \mathbf{b}\right)$$

$A=QR, R=Q_n\dots Q_1A, \vec{x}=R^{-1}(Q^T\vec{b}), Q_i\perp$
Well defined even for singular (or zero-valued) matrices (relies on isometries), more expensive than LU, solve tall linear system giving least squares solution without using normal equations
2x more expensive than LU but extremely well-behaved
Solve : mult. b by Q^T n^2; △ solve with R n^2

```
Q, R = scypy.linalg.qr(A,
↪  mode="economic")
x = scypy.linalg.solve_triangular(R,
↪  Q.T @ b, lower=False)
```

Gram-Schmidt orthogonalisation :
$\hat{y}:=\vec{y}-\frac{\vec{x}\cdot\vec{y}}{\vec{x}\cdot\vec{x}}\vec{x}$, (then $\vec{x}\cdot\hat{y}=0$)
$\langle\vec{x},\vec{y}\rangle=\vec{x}\cdot\vec{y}:=\sum_{i=1}^{n}x_iy_i$
$\langle f,g\rangle:=\int_a^b f(x)g(x)\,dx$

3.4 Eigenvalues Decomposition

$i\in\{1,\dots,n\}, A\vec{v}_i=\lambda_i\vec{v_i}$
$A\in\mathbb{R}^{n\times n}, \lambda_i\in\mathbb{R}, \vec{v}_i\in\mathbb{R}^n, \|\vec{v}_i\|=1$
λ_i : eigenvalue, $\vec{v}_i$: eigenvector
Eigenvectors : left $\vec{y}^TA=\lambda\vec{y}^T$, right $A\vec{x}=\lambda\vec{x}$
Finding eigenvalues : $\det(A-\lambda I)=0$
`λ, V = scipy.linalg.eig(A)`
For non-singular matrices A :
Exactly n distinct (eigenvalue, eigenvector)
Eigenvectors form a (not necessarily orthogonal) basis for $\mathbb{R}^n$
Matrix with $\vec{v}_i$ as its columns has rank n
Decomposition : $A=V\Lambda V^{-1}$, Λ diag. s.t. $a_{ii}=\lambda_i$
$A^{-1}=V\Lambda^{-1}V^{-1}, A^k=V\Lambda^kV^{-1}$
Non-linear operation : approximate result, no guarantees on computation time
Power iterations : as $k\to\infty, {\lambda_1}^k\gg\lambda_{i\neq 1}$ (λ_i are sorted in decreasing order), iteratively apply A to $\vec{x}$ (normalize) converges to $\vec{v}_1$, Fail : if initial guess $\vec{x}_0$ is $\perp$ to eigenvector or in $\mathrm{Ker(A)}$, 2 largest eigenvalues are close to each other, overflows (if not normalizing at each iteration)
Inverse iteration : iteratively apply A^{-1} to $\vec{x}$ (normalize) converges to $\vec{v_n}$ (associated with smallest eigenvalue λ_n), $\frac{1}{\lambda_n}$ dominates as $k\to\infty$, compute eigenvector associated with eigenvalue λ by running inverse iteration on the matrix $A'=A-\lambda I$

Dimensionality reduction : maximize $\vec{v}^TXX^T\vec{v}$
Solving ODEs : $\vec{x}'(t)=A\vec{x}(t), \vec{x}(t)=e^{At}\vec{x}(0), e^{At}=Ve^{\Lambda t}V^{-1}$, where e^{Λ} diag. s.t. $(e^{\Lambda})_{ii}=e^{\lambda_i}$
Problems : complex arithmetic required, methods can be unstable if eigenvalues not well separated, only for square A

3.5 Singular Value Decomposition (SVD)

Good accuracy if system ill-conditioned, slower than LU & QR
Expensive to compute but : stable computation, for any matrix, results in orthogonal and diagonal matrices, impeccable numerical properties, no complex arithmetic

$$\boxed{A}=\boxed{U}\boxed{\Sigma}\boxed{V^T}$$

$V=(\vec{v}_1\quad\dots\quad\vec{v}_3), U=(\vec{u}_1\quad\dots\quad\vec{u}_n)$
Σ diag. s.t. $\Sigma_{ii}=\sigma_i, \sigma_i$ are positive and sorted in decreasing order
U,V are orthogonal matrices ($P^T=P^{-1}$)
$A:m\times n, U:m\times m, \Sigma:m\times n, V^T:n\times n$
$A=\sum_{i=1}^{n}\sigma_i\vec{u}_i\vec{v}_i^T, A^{-1}=V\Sigma^{-1}U^T$
`U, Σ, V = scipy.linalg.svd(A)`
Solve : mult. by U^T n^2, mult. Σ^{-1} n, mult. V n^2
$\|A\|_2=\sigma_1, \mathrm{cond}(A)=\frac{\sigma_1}{\sigma_n}$
$\|A^{-1}\|=\frac{1}{\sigma_n}$
Outer product (rank 1 matrix) : $\vec{u}\vec{v}^T$
SVD Row vector $\vec{x}$: $1\|\vec{x}\|\frac{\vec{x}}{\|\vec{x}\|}$
SVD Column vector $\vec{x}$: $\frac{\vec{x}}{\|\vec{x}\|}\|\vec{x}\|1$

Frobenius norm : $\|A\|_F^2:=\sum_{i,j}A_{ij}^2$
$A':=A$ only keeping the k largest σ
Eckart-Young theorem : A' minimizes both $\|A-A'\|_2$ and $\|A-A'\|_F$ among all matrices A' shaped with rank k
Low-rank approximation : $A\approx\sum_{i=1}^{k_{\max}}\sigma_i\vec{u}_i\vec{v}_i^T$, best possible approximation using outer products
Pseudoinverse : $A^+=\sum_{i=1}^{\min\{n,m\}}\vec{v}_i\vec{u}_i^T\begin{cases}\frac{1}{\sigma_i} & ,\sigma_i\neq 0\\ 0 & ,\sigma_i=0\end{cases}$
Identical to inverse for full-rank matrices, least-squares solution for over-constrained problems, minimum-norm solution for un(der)constrained problems ($\mathrm{argmin}_x\|x\|^2$)
Regularization : explosion due to small σ_i, $A^{-1}=V\Sigma^{-1}U^T=\sum_{i=1}^{n}\frac{1}{\sigma_i}\vec{v}_i\vec{u}_i^T$

4 Nonlinear problems

Root-finding : x such that $f(x)=0$
Intermediate value : f continuous, $f(x_0)=y_0, f(x_1)=y_1$ then $\forall y\in[y_0,y_1]\exists x\in[x_0,x_1]$: $f(x)=y$
Stopping criteria : $|f(m)|<\varepsilon_1$ or $|l-r|<\varepsilon_2$

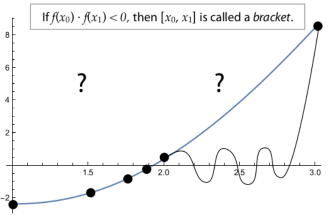
E_k : the error after k iterations
o : order of convergence (how quickly algorithm converges, 1 linear, 2 quadratic)
r : rate of convergence (distinguishes convergence speed of algorithms with same order)
$\lim_{k\to\infty}\frac{E_{k+1}}{E_k^o}=r$

4.1 Function spaces

Continuous : $f(x)\to f(y)$ as $x\to y$
Lipschitz continuous : $\exists c\in\mathbb{R}:|f(x)-f(y)|\le c|x-y|$
Differentiable : $\forall x\in D, f'(x)$ exists
C^k differentiable : $\frac{\partial^i f}{\partial x^i}\in C^i, i\in\{1,\dots,k\}$

4.2 Bisection method

Slow, guaranteed to work, requires continuity + bracket Only function sign is relevant.
$o=1$ and $r=\frac{1}{2}$



```
def bisect(f, l, r, eps1, eps2):
    m = l + (r - l) / 2
    if abs(f(m)) < eps1 or abs(l -
    ↪  r) < eps2:
        return m
    if f(l) * f(m) < 0:
        return bisect(f, l, m, eps1,
        ↪  eps2)
    return bisect(f, m, r, eps1,
    ↪  eps2)
```

4.3 Newton's method

$f(x+h)\approx f(x)+f'(x)\cdot h$
Repeatedly do : $x_k=x_{k-1}-\frac{f(x_{k-1})}{f'(x_{k-1})}$
Extremely fast, diverges when derivative don't capture the function's behavior, requires derivative
Newton-Bisection method (combinaison) : Extremely fast, guaranteed to work, requires continuity + bracket + derivatives, derivative can become small (explosion)
N-D (multivariate) case :
$\vec{f}(\vec{x}+\vec{h})\approx\vec{f}(\vec{x}_{k-1})+\nabla\vec{f}(\vec{x}_{k-1})\vec{h}$
$\vec{x}_k=\vec{x}_{k-1}-[\nabla\vec{f}(\vec{x}_{k-1})]^{-1}\vec{f}(\vec{x}_{k-1})$
No hybrid newton-bisection in N-D, linear system solve could fail, computation of Jacobian and inverse takes n^3 time

4.4 Broyden's method

Faster Newton-like for multivar. roots, don't compute Jacobian, start with a guess and improve over time : $J_k = J_{k+1} + \Delta J$

$\Delta J = \frac{\Delta\vec{f} - J_{k-1}\Delta\vec{x}}{\|\Delta\vec{x}\|}\Delta\vec{x}^T$ (rank-1 matrix)

Fast Jacobian approximation, invert Jacobian still slow (n^3) : Sherman-Morrison formula for fast Jacobian inverse $(A + \vec{u}\vec{v}^T)^{-1} = A^{-1} - \frac{A^{-1}\vec{u}\vec{v}^T A^{-1}}{1+\vec{v}^T A^{-1}\vec{u}}$

Evaluate $\vec{f}(\vec{x}_k)$, set J_{k+1} using $J_k, \vec{f}(\vec{x}_{k+1}), \vec{f}(\vec{x}_k)$, update J_k^{-1} using Sherman-Morrison, approximate newton step

5 Automatic differentiation

5.1 1 dimension

Unimodal function : $\exists m$ such that f monotonically decreasing (increasing) for $x \geq m$ ($x \leq m$)

Golden section search : (like bissection) using optimal sampling

```
p = 2 / (sqrt(5) + 1) # φ^-1
x1 = r - (r - l) * p
x2 = l + (r - l) * p
if f(x1) < f(x2): r = x2
else: l = x1
# return (l + r) / 2
```

5.2 N dimensions

Gradient descent : $\vec{x}_{k+1} = \vec{x}_k - \alpha\nabla f(\vec{x}_k)$ α : step size (chosen in each iteration by solving 1D problem)

Taylor approximation : $f(\vec{x} + \vec{h}) \approx f(\vec{x}) + \nabla f(\vec{x}) \cdot h + \frac{1}{2}\vec{h}^T H_f(\vec{x})\vec{h}$

Newton's method : min as root finding using $\nabla f(\vec{x}) = 0$ (could converge to any other local extremum), $\vec{x}_k = \vec{x}_{k-1} - H_f^{-1}(\vec{x}_{k-1})\nabla f(\vec{x}_{k-1})$

5.3 Automatic differentiation

Store value and derivate at same time (forward mode) :

Addition : $[v_1, \partial_x v_1] + [v_2, \partial_x v_2] = [v_1 + v_2, \partial_x v_1 + \partial_x v_2]$

Multiplication : $[v_1, \partial_x v_1] \cdot [v_2, \partial_x v_2] = [v_1 \cdot v_2, v_1\partial_x v_2 + v_2\partial_x v_1]$

Reverse mode : evaluate chain rule in reverse

Finite diff. : forward $f'(x) \approx \frac{f(x+h)-f(x)}{h}$, centered $\frac{f(x+h)=f(x-h)}{2h}$

6 Neural networks

$\vec{x}$: input, $\vec{w}$ weight, b : neuron bias, h : activation function, n : number of layers (excluding input), D : dataset

Activation functions : $\mathrm{sigmoid}(x) = \frac{1}{1+\exp(-x)}$, $\mathrm{relu}(x) = \max 0, x$

Computation :

neuron : $h\left(\sum_i \vec{w}_i\vec{x}_i + b\right)$

fully connected : $\vec{a} = h(W\vec{x} + \vec{b})$

layered : $a^{(1<i<n)} = h(W^{(i)}\vec{a}^{(i-1)} + \vec{b}^{(i)})$

output : $a^{(n)} = W^{(n)}\vec{a}^{(n-1)} + \vec{b}^{(n)}$

$W\vec{x} + \vec{b} = \widetilde{W}\widetilde{x}, \widetilde{W} = [W||\vec{b}], \widetilde{x} = \begin{bmatrix}\vec{x}\\1\end{bmatrix}$

$nn = h(\vec{W}^{(n)}\dots h(\vec{W}^{(1)}\vec{x}+\vec{b}^{(1)})\dots+\vec{b}^{(n)})$

Training : $\arg\min_W E(D,\theta), E(D,\theta) = \sum_{(x^{(i)},y^{(i)})\in D}(nn(x^{(i)},\theta) - y^{(i)})^2$

Loss function : $\arg\min_\theta E(D,\theta)$ $E(D,\theta) = \sum_{(\vec{x}^{(i)},y^{(i)})\in D} l(nn(\vec{x}^{(i)},\theta), y^{(i)})$

quadratic loss : $l_2(y,l) = (y-l)^2$
absolute loss : $l_1(y,l) = |y-l|$

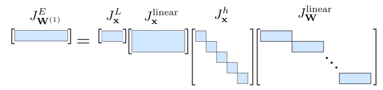
Stochastic gradinet descent : α : learning rate, $\mathbb{B}$: minibatch (training subsets), start normal uniform random $W_{i,j}$ around 0 of variance σ^2 ($N(0,\sigma^2)$), $\nabla_\theta E(B,\theta)$, update weights $\theta^{(t+1)} = \theta^{(t)} - \alpha\nabla_\theta E(\mathbb{B},\theta)$

Regression : model a continuous function mapping input points to points in the output space

Classification : assigns input points to discrete classes (probabilistic)

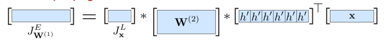
Backprop : reverse mode differentiation, Jacobian not computed

Jacobian formulation



$J^E_{\mathbf{W}^{(1)}}$
$J^L_{\mathbf{x}}$
$J^{\text{linear}}_{\mathbf{x}}$
$J^h_{\mathbf{x}}$
$J^{\text{linear}}_{\mathbf{W}}$

Compact backpropagation



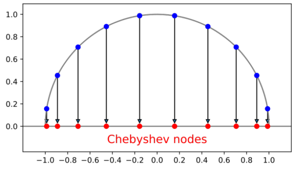
7 Interpolation

7.1 Chebyshev nodes

Error for polynomial passing through points $x_1,\dots,x_n : E(x) = \frac{f^{(n)}(\xi)}{n!}\prod_{i=1}^n(x-x_i)$

Minimize : $\max\limits_{x\in[-1,1]}\left|\prod_{i=1}^n(x-x_i)\right|$

Resulting positions : $x_{1\leq i\leq n} = \cos\left(\frac{2i-1}{2n}\pi\right)$

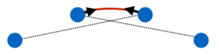


Points resulting in less error, especially on edges

7.2 Splines

Very local point of view, no explosions, split function and find one function for each pair of points

Catmull-Rom : take derivative estimation using straight line from point before/after



Cubic splines : solve linear system using $f(x) = ax^3+bx^2+cx+d$, $(f'(x) = 3ax^2+bx^2+c)$, where $f(x_0), f(x_1), f'(x_0), f'(x_1)$ known

```
f = scipy.interpolate.interp1d(X, Y,
  kind=3)
```

7.3 Real basis function

Used to assign distance weighted influence to each data point

Gaussian : $\phi(r) = e^{-(\varepsilon r)^2}$

Inverse quadratic : $\phi(r) = \frac{1}{1+(\varepsilon r)^2}$

Inverse multiquadric : $\phi(r) = \frac{1}{\sqrt{1+(\varepsilon r)^2}}$

Given n evaluations $f(\vec{x}_i)$ at positions $\vec{x}_i$:

Interpolant : $g(\vec{x}) = \sum_{i=1}^n w_i\phi(\|\vec{x} - \vec{x}_i\|)f(\vec{x}_i)$ find w_i such that $g(\vec{x}_i) = f(\vec{x}_i)$ (linear system)

```
f = scipy.interpolate.Rbf(X, Y, Z,
  epsilon=2)
```

8 Numerical approximations

Given f, design an approximation $\hat{f}$ such that $||\hat{f} - f||$ is very small

Remez algorithm : uniform max error, numerical procedure
Find local max/min using golden section search, find new polynomial using equioscillation theorem

$$\begin{bmatrix}1 & x_0 & x_0^2 & x_0^3 & \cdots & +1\\ 1 & x_1 & x_1^2 & x_1^3 & \cdots & -1\\ \vdots & & & & & \vdots\\ 1 & x_n & x_n^2 & x_n^3 & \cdots & \pm1\end{bmatrix}\begin{bmatrix}a_0\\ \vdots\\ a_n\\ E\end{bmatrix} = \begin{bmatrix}f(x_0)\\ \vdots\\ f(x_n)\\ f(x_{n+1})\end{bmatrix}$$

where x_i are found extrema

9 Integration

Given function evaluations $f(x_i)$ at nodes $x_i \in [a,b], \Delta x = (b-a)$

Riemann : $\int_a^b f(x)\,dx = \lim\limits_{x\to0}\sum_k f(x_k)\Delta x \approx \sum_{k=1}^n f(x_k)\Delta x$

Composite quadrature midpoint : $\int_a^b f(x)\,dx \approx \sum_{i=1}^k f(\frac{x_{i+1}+x_i}{2})\Delta x$, not good if function varies a lot on interval

Trapezoid rule : $\int_a^b f(x)\,dx \approx (b-a)\frac{f(a)+f(b)}{2}$, good if function close to linear

Composique quadrature trapezoid : $\int_a^b f(x)\,dx \approx \sum_{i=1}^k(\frac{f(x_i)+f(x_{i+1})}{2})\Delta x = \Delta x(\frac{1}{2}f(a)+f(x_1)+\ldots+f(x_{k-1})+\frac{1}{2}f(b))$

Simpson rule : $\int_a^b f(x)\,dx \approx \frac{b-a}{6}(f(a) + 4f(\frac{a+b}{2}) + f(b))$, exact for quadratic polynomial (n must be odd)

Composite quadrature Simpson : n is odd, $\int_a^b f(x)\,dx = \frac{\Delta x}{3}(f(a) + 2\sum_{i=1}^{n-2-1} f(x_{2i}) + 4\sum_{i=1}^{n/2} f(x_{2i-1}) + f(b))$

Adaptive quadrature : if absolute difference between trapezoid and Simpson $>$ $\epsilon\cdot$trapezoid then add midpoint (new evaluation point), divide and conquer, else return Simpson

Newton-cotes : n uniformly spaced points, exacte for polynomials of degree n, $Q[f] := \sum_i w_i f(x_i)$ (linear system, solve for w_i)

10 Monte Carlo Methods

Fight curse of dimensionality, extremely general approach, easy to implement, use of randomness to simulate possible realizations of complex process, very slow convergence, very bad method, used only when alternatives are worse, effective for high-dimensional functions

Averaging many samples : $E[\frac{1}{N}\sum_{i=1}^N f(\vec{X}_i)] = \frac{1}{N}\sum_{i=1}^N\int_\Omega f(\vec{x})p_{\vec{x}_i}(x)dx$

Integration : $\int_\Omega f(\vec{x})\,d\vec{x} \approx \frac{1}{N}\sum_{i=1}^N f(\vec{X}_i)$, $\vec{X}_i$ is random variable, distribution is centered around correct answer, algorithm will give random answer so always some error

Intergration error : $\mathrm{Var}[\frac{1}{N}\sum_{i=1}^N f(\vec{X}_i)] = \frac{1}{N}\mathrm{Var}[f(\vec{X}_i)]$, where $\mathrm{Var}[f(\vec{X}_i)]$ is constant, error reduction proportional to $\frac{1}{\sqrt{N}}$, error $\approx \frac{1}{\sqrt{N}}\mathrm{Var}[f(\vec{X}_i)]$

Importance of sampling : $E[\frac{1}{N}\sum_{i=1}^N\frac{f(\vec{X}_i)}{p_{\vec{X}}(\vec{X}_i)}] = \int_\Omega f(\vec{x})\,d\vec{x}$, $p_{\vec{X}}(\vec{X}_i) > 0\,\forall i$ with $f(\vec{X}_i) \neq 0$, non-uniform sampling (according to distribution $p_{\vec{x}}$)

Uniform variates : realization of uniformly distributed variable in $[0.0, 1.0]$, sequence should pass statistical tests of randomness & have long period & efficient to compute & requires little storage & repeatability = always produce same sequence

```
def mc_unif(f, a, b, n):
  var=random(size=n)*(b-a)+a#np
  pdfs=1/(b-a)
  return np.sum(f(var)/pdfs)/n
```

Rejection sampling : from uniform distribution rejects larger than $p(\vec{x})$

Inversion method : generate samples according to probability density $p_X(x)$, compute CDF (cumulative distribution function) $P_X(x) = \int_0^x p_X(x')\,dx'$, compute inverse $P_X^{-1}(y)$, generate uniform variate ξ_i and evaluate $P_x^{-1}(\xi_i)$, result will have desired distribution

General recipe : $x_i := P_X^{-1}(\xi_i),\ (i = 1,\dots n)$, over N uniform variates ξ_i, average Monte Carlo integrand $\frac{f(x_i)}{p_X(x_i)}$

11 Examples

11.1 Kinds of numerical errors

trunc, cancel | overflow, underflow

```
res=f64(<frac. value btw 0 and 1>)
for i in range(1000, 0, -1):
  sign = 1 if (i % 2 == 0) else -1
  res += sign*(np.pi*f64(i)**2)
```

overflow | trunc., cancel., underflow

```
res=f64(<frac. value btw 0 and 1>)
for i in range(1000):
  res+=np.exp(i)/(np.float64(i)**2)
```

trunc., cancel., over/under-flow

```
res = f64(<small pow. of 2>)
for i in range(10):
  res+=res if i%2==0 else:res*=4
res /= 1024
```

Distributive law : $a\cdot(b+c) \neq a\cdot b + a\cdot c$. If $b+c$ overflows and $a = 0$, LHS is NaN and RHS ∞

11.2 Breaking things

Overflow (*), Underflow (/)

```
a = 1.0#to return
for i in range(1000): a *= 10.0#/=
```

NaN : `np.array(1.0)/arr(0.0)*arr(0.0)`

Non-terminating loop : $N > 0$ finite FP value, `i=0.0;while i<N:i+=1`

11.3 Performance implications

```
#for j in range(n):for i in rag(n):
for i in range(n):for j in rag(n):
  if x[i]>0:A[i,j]+=1.0
```

2-nd line have sequential memory access, cache hierarchy, better runtime

Branching predictions in modern processors, stables branches are faster

11.4 Linear systems

Underdetermined SVD : Find $\arg\min_x \|x\|^2$

```
def solve_underdetermined(A,b):
  m, n = A.shape,
  x,y = np.zeros(n), []
  U, S, Vt = svd(A)
  for i in range(n):if i<m:
    x+=(U[:,i]@b)/S[i,i]*Vt[i,:]
    else:y.append(Vt[i,:])
```

Newton's method : Find minima of $\mathrm{sinc}(x) = \frac{\sin x}{x}$: $\mathrm{sinc}', \mathrm{sinc}''$:

```
def newton_sinc(x):#return x
  for i in range(10):
    x-=sinc_p(x)/sinc_pp(x)
```

Diverge when dividing by very small second derivative, converge to other local extremum

meth./Eval./conv./deriv./sure : sec./1/2/N/N, newt./2/1/Y/N,bisc./1/3/N/Y

$A(x+m) = b+n, Am = n, m = A^{-1}n$

11.5 ODE

```
def solve_ode(x0, a0, beta, t):
  A = np.array([[c1,c2],[c3,c4]])
  X0 = np.array([x0,a0])
  L,V = la.eig(A)
  E = V @ np.diag(np.exp(L * t)) @
  ↪  la.inv(V)#return (E @ X0)[0]
```